UNIVERSITA' DEGLI STUDI DI PISA

FACOLTA' DI INGEGNERIA

Corso di Laurea Specialistica in Ingegneria Informatica

TESI DI LAUREA

Air Traffic Management:

Analisi e Implementazione del protocollo RAPTOR utilizzando OMNeT++

Relatori:

Prof. Cinzia Bernardeschi

Prof. Gianluca Dini

Candidato:

Salvatore Buonaguro

ANNO ACCADEMICO 2009/2010

A mia madre Maria Rosaria,

che questo mio successo possa portarle un po' di quella felicità,

che in questo momento come non mai, merita così tanto.

Indice

Sommario

Scopo di questo lavoro è stato quello di studiare, implementare e simulare alcuni degli algoritmi di consenso (agreement) per sistemi distribuiti tolleranti ai guasti, utilizzando l'ambiente OMNeT++. In particolare è stato analizzato il protocollo RAPTOR, presentato recentemente per la gestione del traffico aereo di tipo distribuito, noto come airborne self-separation.

CAPITOLO 1

INTRODUZIONE

Con il termine **Aircraft Traffic Management** (da qui in avanti ATM) viene indicato l'intero sistema che controlla e gestisce il traffico aereo. Principale caratteristica dell'ATM è l'elevato livello di affidabilità e disponibilità (dependability) richiesto anche in presenza di malfunzionamenti.

L'ATM classico è basato su una pianificazione rigida ed offline, e fa uso di controllori del traffico principalmente di terra chiamati Air Traffic Control (ATCOs, torri di controllo). Per questo tipo di ATM, concettualmente, lo spazio aereo viene diviso in settori, e ciascun settore viene assegnato ad un ATCO, che ne diventa l'autorità centrale. Gli ATCOs sono responsabili del mantenimento della separazione sia orizzontale che verticale dei vari aircraft coinvolti, in modo da assicurare un traffico aereo il più possibile rapido ed ordinato, indicando agli aircraft le direzioni da seguire, e fornendo ai piloti informazioni aggiuntive sulle rotte, come per esempio le condizioni atmosferiche.

Questo tipo di ATM, incentrato sui controllori di terra, fa affidamento sulla capacità dei controllori stessi di gestire tutto il sistema, con pochissima autonomia per piloti e compagnie aeree. Inoltre, essendo per costruzione fortemente centralizzato, presenta problemi di scalabilità nel fronteggiare il sempre più crescente aumento del volume di traffico aereo.

La ricerca, pertanto a partire dai primi anni 2000, si è dedicata allo sviluppo di approcci e soluzioni alternative, che potessero superare i limiti imposti dall'ATM classico come sempre concepito.

I recenti progressi nel campo tecnologico stanno rendendo possibile un ATM di nuova concezione, che prende il nome di ***airborne self-separation***. In questo nuovo ATM la responsabilità della separazione orizzontale e verticale dei vari aerei viene spostata dal controllo terrestre sugli aerei stessi, ed è lasciata in modo sempre maggiore la possibilità per i piloti di scegliere i loro *flight paths,* senza l'intervento diretto di controllori centralizzati.

In questo futuro ***self-separation enviroment,*** i piloti avranno una maggiore responsabilità nel condurre i voli in modo sicuro ed efficiente, e dovranno essere supportati da un sistema automatizzato di decisione che, processando tutte le informazioni disponibili, assisterà il pilota nello scegliere la traiettoria ottimale compatibilmente con il mantenimento della separazione dello spazio aereo fra i vari aerei.

I vantaggi saranno di due tipi:

- Prima di tutto, una maggiore efficienza. Una gestione distribuita del traffico consente infatti a ciascuna compagnia aerea di dare maggiore priorità a determinati parametri e fattori rispetto ad altri, in accordo alle diverse strategie della singola compagnia.
- In secondo luogo, una maggiore scalabilità, che consentirà alle compagnie aeree di aumentare la capacità di traffico e di soddisfarne la sempre più crescente domanda, con una conseguente riduzione dei costi.

Un approccio automatico e distribuito del traffico aereo pone però al tempo stesso, maggiori problematiche di affidabilità e sicurezza. Ogni aircraft infatti fa affidamento su informazioni fornite dagli altri aerei posti nelle vicinanze: **diventa imperativo pertanto, assicurare che queste informazioni siano scambiate in modo affidabile e sicuro.**

Sappiamo che la comunicazione wireless è per sua stessa natura inaffidabile e la presenza di un singolo aircraft difettoso che trasmette informazioni incoerenti o “jam information” può portare a conseguenze impreviste e potenzialmente catastrofiche per la sicurezza del trasporto aereo.

Fondamentale risulta quindi la definizione di algoritmi distribuiti tolleranti ai guasti e l’analisi di questo tipo di protocolli con strumenti di simulazione.

Il lavoro è organizzato come segue:

- Breve descrizione della tecnologia ADS-B che si sta progressivamente diffondendo e che rende possibile un ATM distribuito di nuova concezione (Capitolo 2).
- Introduzione al problema del consenso nei sistemi distribuiti con la descrizione dell’algoritmo dei Generali Bizantini (Capitolo 3).
- Presentazione dell’ambiente OMNeT++ (Capitolo 4).
- Presentazione dello stack di protocolli RAPTOR (Capitolo 5).
- Formalizzazione, implementazione e simulazione di RAPTOR in ambiente OMNeT++ e linguaggio C++ (Capitoli 6-7).
- Implementazione dell’algoritmo dei Generali Bizantini in ambiente OMNeT++ (Capitolo 8).
- Conclusioni e possibili sviluppi futuri (Capitolo 9).

CAPITOLO 2

ADS-B: AUTOMATIC DEPENDENT SURVELLEINCE BROADCAST

In questo capitolo, daremo una panoramica veloce sulla tecnologia ADS-B che sta alla base dell'ATM di nuova concezione, e che sta progressivamente sostituendo i sistemi RADAR in funzione sin dall'immediato dopoguerra in molte zone del pianeta. Si contano ormai numerose installazioni di infrastrutture ADS-B in Australia, USA, Nord Europa, e prestissimo saranno presenti anche in Italia.

Non è ovviamente nostro scopo entrare nei dettagli del funzionamento a livello fisico del segnale e della strumentazione ADS-B, ma solo mostrare le principali caratteristiche del modulo ADS-B che andremo ad implementare sugli aircraft all'interno del modello simulato in ambiente OMNeT++.

2.1 Che cos'è l'ADS-B

L'ADS-B è sostanzialmente una nuova tecnologia che ridefinisce radicalmente il paradigma **Comunicazione-Navigazione-Sorveglianza** *(Communications-Navigation-Surveilleince)*, in modo da permettere ai piloti e ai vari Air Traffic Controller di "vedere" e controllare i mezzi di volo con molta più precisione e su una percentuale della superficie del pianeta nettamente superiore a quanto

avvenuto finora con i sistemi RADAR. L'ADS-B sta già velocemente sostituendo in moltissime aree del pianeta l'infrastruttura RADAR.

Vediamo per prima cosa significa l'acronimo ADS-B:

Automatic – E' sempre in funzione e non richiede l'intervento di un qualche operatore esterno (es., torri di controllo da terra).

Dependent – Dipende e fa affidamento sull'accuratezza del segnale GNSS (Global Navigation Satellite System) per tutto ciò che riguarda le coordinate e i dati di posizione degli aircraft.

Surveillance – Fornisce e mette a disposizione "servizi di sorveglianza", come e più dei sistemi RADAR.

Broadcast – Spedisce di continuo in broadcast la posizione e altri dati utili a tutti gli aircraft o alle Stazioni di Terra equipaggiate con sistemi ADS-B.

2.2 Come funziona l'ADS-B

A differenza del sistema radar che funziona inviando onde per mezzo di antenne terrestri fisse e reinterpretando i segnali riflessi, l'ADS-B sfrutta il sistema e la tecnologia di Global Navigation Satellite System, il *GNSS*, e si appoggia essenzialmente su semplici link di comunicazione broadcast (dati appunto dal sistema satellitare) come componenti principali della propria infrastruttura.

Ancora, a differenza del sistema radar, l'accuratezza e la precisione dell'ADS-B non si degradano seriamente all'aumentare delle distanze, al variare delle condizione atmosferiche o in funzione

dell'altitudine dell'aircraft. Infine, gli intervalli di aggiornamento del segnale non sono vincolati e dipendenti dalla velocità di rotazione e dall'affidabilità di strumenti meccanici come le antenne.

Nella pratica, un aircraft equipaggiato con strumentazione ADS-B, sfrutta il segnale di un ricevitore GNSS (es., GPS) per ottenere le sue precise coordinate all'interno della costellazione GNSS e poi le combina con tutte le altre informazioni di utilità, quali velocità, accelerazione, altitudine, ID di volo etc. Queste informazioni vengono spedite simultaneamente in broadcast a tutti gli altri aircraft, stazioni di terra o satelliti dotati di strumentazione ADS-B che rilanciano a loro volta queste informazioni, in tempo reale, ai centri di controllo del traffico, come mostrato schematicamente in figura:

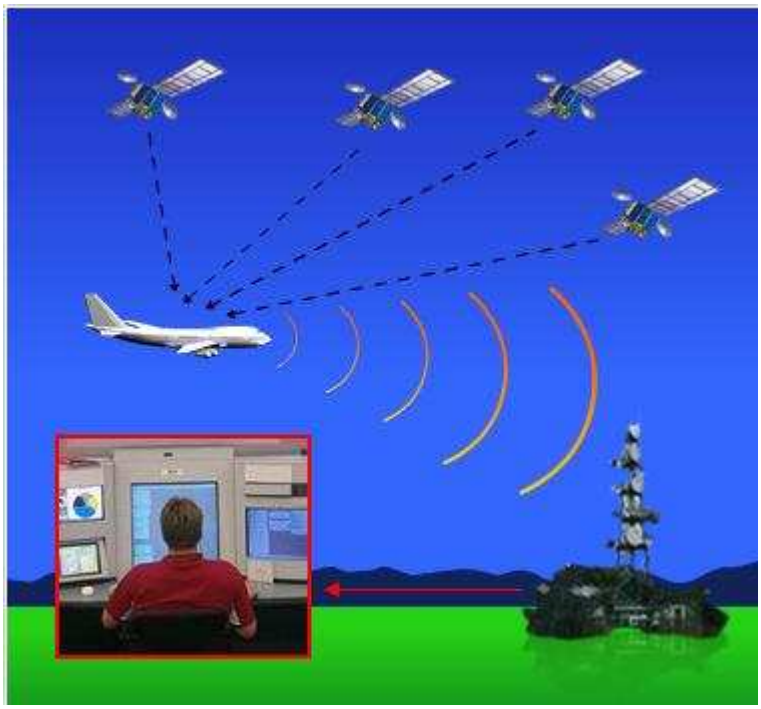


Figura 2.1: semplice schema di funzionamento della tecnologia ADS-B

Uno dei principali fattori che ha portato progressivamente alla sostituzione (ancora in corso) del consolidato sistema RADAR in uso fin dall'immediato dopoguerra, è stato sicuramente il basso costo che l'infrastruttura di questa nuova tecnologia richiede.

Per garantire la stessa percentuale di copertura di un determinato territorio l'ADS-B richiede da un decimo a un ventesimo dei costi di un sistema RADAR. Inoltre i bassissimi requisiti di potenza del segnale richiesti rispetto ai sistemi RADAR, hanno permesso l'installazione di *ground-station* ADS-B negli angoli e nelle zone più remote del pianeta, permettendo una più alta copertura della superficie terrestre.

All'interno dell'URL di riferimento del progetto ADS-B **[9]**, è possibile trovare alcune immagini che mostrano l'installazione di infrastrutture di terra in zone ad elevatissima altitudine, o dalle condizioni atmosferiche proibitive, come in Alaska, ai confini della zona polare.

2.3 Vantaggi e benefici della tecnologia ADS-B

L'ADS-B è concepito per incrementare sia la sicurezza che l'efficienza del volume di traffico. L'ADS-B è *automatico*: non richiede alcun intervento da parte di piloti, controllori o altri operatori esterni.

Ricapitolando, i principali benefici portati dalla tecnologia ADS-B sono:

- Maggiore accuratezza e precisione, dovuto al tipo di infrastruttura utilizzata, il sistema satellitare, e quindi alla totale indipendenza dalla velocità di rotazione delle antenne meccaniche. Gli intervalli di aggiornamento del segnale sono notevolmente più bassi.
- Minore degrado del segnale in condizioni atmosferiche difficili, o in casi di altitudine elevata dell'aircraft
- Una separazione dello spazio aereo tra i vari aircraft ridotta.

- Minori requisiti di potenza del segnale richiesti: questo permette di raggiungere una copertura del segnale molto maggiore.
- Costi di installazione notevolmente più bassi rispetto ai sistemi RADAR a parità di copertura territoriale.
- Possibilità di ottenere una copertura della superficie terrestre notevolmente più grande.

CAPITOLO 3

IL PROBLEMA DEL CONSENSO: Algoritmo dei Generali Bizantini

Il consenso (agreement) è uno dei problemi fondamentali nei sistemi distribuiti soggetti a guasti, in cui sia necessario garantire un elevato grado di affidabiltà. Gli algoritmi di consenso fanno si che i componenti ben funzionanti del sistema si mettano d'accordo su una decisione comune in presenza di componenti guasti, che possono volutamente (malicious components) oppure casualmente inviare informazioni diverse a diverse parti del sistema.

Gli algoritmi di consenso presentati in letteratura sono vari ed il problema è un argomento di ricerca attuale.

Nel 1982 Lamport, Shostak e Pease studiarono il problema del consenso **[5]**, introducendo la metafora dei Generali Bizantini e proponendo un algoritmo per sistemi sincroni e per un modello di guasti che non pone vincoli al comportamento di componenti guasti (appunto detti guasti Bizantini).
In questo capitolo riportiamo un'illustrazione completa del problema e dell'algoritmo presentato in **[5]**. L'implementazione dell'algoritmo in OMNeT++ è descritta in un capitolo successivo.

3.1 I Generali Bizantini

Immaginiamo un certo di numero di divisioni dell' esercito Bizantino accampate attorno a una città nemica, ciascuna divisione capeggiata da un Generale.

Dopo aver osservato il nemico, i Generali devono ora mettersi d'accordo su un piano comune: tuttavia alcuni di essi potrebbero essere Traditori e cercare di impedire che i Generali Leali raggiungano una decisione comune. Inoltre non facciamo assunzioni sulla possibilità di comportamento dei generali Traditori.

Un generale Traditore potrebbe inviare informazioni differenti a generali diversi o comunque esibire un comportamento arbitrario.

Un algoritmo di consenso è un algoritmo che garantisce le seguenti proprietà:

1. **tutti i Generali Leali decidono lo stesso piano d'azione comune**

2. **un "piccolo" numero di Generali Traditori (definiremo in seguito con precisione quanto vale questo "piccolo") non deve portare i Generali Leali ad accordarsi su un piano d'azione sbagliato.**

Quest' ultima condizione è difficile da formalizzare: si tratta di definire precisamente cos' è un "piano d' azione sbagliato", e questo dipende ovviamente dal sistema, dall' ambiente e dalle variabili considerate.

Ci limiteremo per il momento a considerare il caso di agreement di tipo binario. Assumiamo che le uniche possibilità di scelta per un Generale siano quelle di "attacco" e "ritirata" e che i Generali debbano accordarsi su una di queste due azioni. In questa situazione, un piano d'azione sbagliato

corrisponde alla decisione “ritirata” nel caso in cui la maggioranza dei Generali Leali abbia proposto “attacco”.

Mostreremo successivamente anche l’algoritmo per gestire casi con più valori (multivalued) e non più binari.

3.1.2 Enunciato del problema

Consideriamo quindi per il momento semplicemente come i generali debbano giungere ad una decisione. Ogni generale dopo aver osservato il nemico deve comunicare agli altri la propria decisione: sia *v(i)* la decisione comunicata dal generale i-esimo. Ciascun Generale dovrà poi usare un qualche metodo per ottenere un singolo piano, “attacco” o “ritirata”, dalle informazioni *v(1), v(2), v(3)…v(n)* ricevute.

Nel seguito useremo semplicemente il termine valore con il significato di decisione proposta quando è chiaro dal contesto.

La condizione **1** precedentemente enunciata è soddisfatta se tutti i generali utilizzano lo stesso metodo per decidere a partire da informazioni identiche *v(1), v(2), v(3)…v(n)*.

La condizione **2** è soddisfatta utilizzando un metodo che potrebbe essere, ad esempio, decidere in base alla maggioranza delle informazioni ricevute.

In questo caso un piccolo numero di generali Traditori potrebbe influire sulla decisione dei generali Leali se e solo se questi ultimi sono quasi equamente divisi tra le due possibilità, “attacco” e “ritirata”, caso nel quale nessuna delle due possibili decisioni può comunque essere definita sbagliata.

I Generali devono avere a disposizione un metodo tramite il quale poter comunicare la propria decisone: assumiamo che i generali inviino direttamente la propria decisione proposta *v(i)* a tutti gli altri Generali. Affinché la condizione **1** sia soddisfatta, dobbiamo essere sicuri che tutti i Generali prendano la decisione a partire dallo stesso array di valori *v(1),v(2)…..v(n)*.

Dal punto di vista di un singolo *v(i),* deve valere che:

- **Una qualsiasi coppia di Generali Leali deve usare lo stesso valore per *v(i).***

Poichè non facciamo assunzioni sul comportamento dei generali Traditori (un generale Traditore potrebbe inviare valori differenti a diversi generali per cercare di depistarli), un Generale non deve usare il valore di decisione ricevuto direttamente dal Generale i-esimo, perché il Generale *i* potrebbe essere un Traditore e avere inviato valori differenti e contraddittori in giro.

Al tempo stesso, affinché la condizione **2** sia soddisfatta, deve valore che:

- **se l' i-esimo Generale è Leale, allora il valore che spedisce deve essere usato da tutti i Generali Leali come il valore di *v(i).***

Spostiamo ora la nostra analisi su come i generali possano mettersi d'accordo su un comando ("attacco" o "ritirata") inviato da un generale.

Riscriviamo cioè il problema nei termini di un Generale comandante (*il sorgente*) che deve inviare un ordine ai suoi Luogotenenti, ottenendo l'enunciato del **Problema dei Generali Bizantini.**

Problema dei Generali Bizantini:

Un Generale comandante deve inviare un ordine ai suoi (n-1) Luogotenenti in modo tale che:

- **Tutti i Luogotenenti ottengano lo stesso ordine**

- **Se il Generale comandante è leale, allora tutti i Luogotenenti leali obbediscono all' ordine impartito.**

Per risolvere il nostro problema originario, dobbiamo risolvere n istanze del Problema dei Generali Bizantini, dove nell'istanza i-esima, l' i-esimo Generale è il Generale Comandante e spedisce la sua decisione, considerando tutti gli altri generali in quell' istante come i suoi Luogotenenti.

3.1.3 Un primo esempio: 3 Generali, un Traditore

Consideriamo come primo esempio, uno scenario in cui abbiamo il Generale comandante (processo sorgente) e due Luogotenenti che ricevono gli ordini da quest' ultimo. Generale e Luogotenenti leali devono accordarsi alla fine sull'ordine a cui obbedire. Nel seguito indicheremo il processo sorgente con S e i due Luogotenenti con A e B.

Assumiamo che il processo sorgente S invii agli altri due processi A e B il valore 1 per "attacco" e il valore 0 per "ritirata".

Una possibile soluzione è la seguente: i due processi si consultano tra loro per sapere che valori hanno ricevuto dal processo sorgente S.

Ogni processo dice agli altri cosa ha ricevuto dal processo sorgente S, e poi ciascun processo deciderà il valore corretto semplicemente prendendo la maggioranza dei valori ricevuti.

Nel seguito mostriamo che, con un processo guasto, in questa situazione non è possibile raggiungere il consenso.

Per un Luogotenente, se riceve informazioni diverse dal comandante e dall'altro Luogotenente, è facile giungere alla conclusione che qualcuno sta mentendo, mentre sapere chi due abbia mentito e, di conseguenza, quale sia il valore corretto non è possibile **[5]**.

A fronte di due valori diversi, un Luogotenente può decidere:

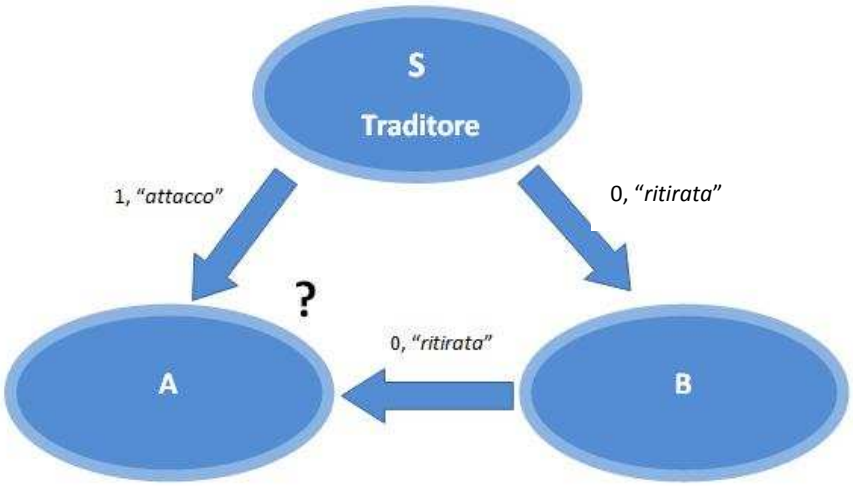
D1: il valore corretto è quello inviato dal generale comandante

D2: il valore corretto è quello inviato dal luogotenente

Ciascuna delle due scelte si rivela però sbagliata in un caso particolare.

Si osservino a i due scenari riportati di seguito.

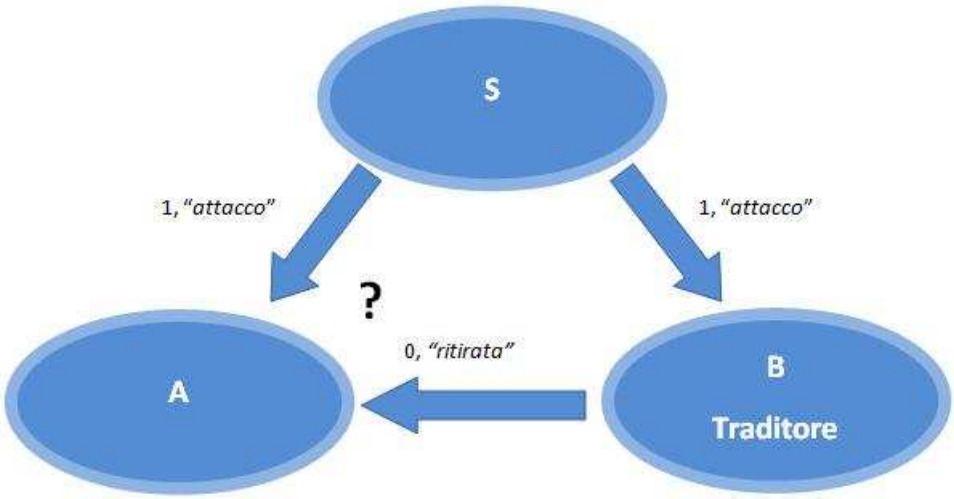
Scenario 1. Supponiamo che il processo Generale comandante sia un Traditore e che invii il comando di attacco al Luogotenente A (1, "attacco") e il comando di ritirata al Luogotenente B (0, "ritirata").



In questo caso, la scelta D1 è sbagliata e la scelta corretta è D2.

Consideriamo ora il seguente scenario.

Scenario 2. Supponiamo che il processo B sia un Luogotenente Traditore e che, dopo aver ricevuto il comando di attacco dal comandante (1, “attacco”), invii (0, “ritirata”) al Luogotenente A.



In questo caso, la scelta D1 è corretta, mentre la scelta D2 è sbagliata.

Come si può notare raffrontando le due figure, il processo A, localmente e singolarmente, non può in alcun modo sapere chi sta mentendo, se il processo Sorgente come nel primo scenario o il processo B, come nel secondo scenario.

Il problema deriva dal fatto che un traditore può portare la massima confusione, inducendo metà dei processi ad attaccare e metà a ritirarsi.

In **[5]**, è stato dimostrato che, dati *N=numero totale di processi* e *M=numero di processi traditori,* la soluzione al problema dei Generali Bizantini è possibile sotto la condizione:

$$N>=3M+1.$$

In altre parole, con un processo traditore, come negli esempi sopra riportati, sono necessari almeno 4 processi in totale perché il problema del consenso sia risolvibile.

3.2 L' algoritmo di Lamport, Pease e Shostak

Andiamo ora ad illustrare la soluzione al problema dei Generali Bizantini pubblicata nel 1982 in **[5]** da Lamport, Pease e Shostak.

L'algoritmo si basa su messaggi scambiati "a voce", *per via orale (oral messages)*.

Questo vuol dire che un Luogotenente può mentire sul valore ricevuto dal generale comandante, quando invia questa informazione agli altri Luogotenenti.

L'algoritmo é valido sotto le seguenti assunzioni:

A1. Ogni messaggio spedito, viene consegnato correttamente

A2. Il receiver del messaggio, sa da chi questo è stato mandato

A3. L'assenza di un messaggio può essere rilevata.

Le prime due assunzioni prevengono il caso in cui un Traditore voglia interferire con la comunicazione tra due altri Generali qualsiasi. L'assunzione A3 esclude il caso che un Traditore voglia portare confusione semplicemente evitando di inviare i propri messaggi.

L'algoritmo *Oral Messagge* è definito parametrico rispetto al numero massimo *M* di processi guasti: *OM(M),* con *M* intero non negativo.

L'algoritmo *OM(M)* si appoggia inoltre sulla funzione **majority($v_1, v_2, \ldots v_n$)** definita nel modo seguente: se la maggioranza dei valori v_i è uguale a *v*, allora **majority($v_1, v_2, \ldots v_n$)** restituisce v. Se non esiste una maggioranza, allora la funzione restituisce un valore di default.

L'algoritmo è definito in modo ricorsivo come segue:

Algoritmo OM(0)

1. Il Generale manda il suo valore a ogni luogotenente.
2. Ogni luogotenente usa il valore che riceve dal generale.

Algoritmo OM(M), M > 0

1. Il Generale manda il suo valore (ordine) a ogni luogotenente.
2. Per ogni i, sia v_i il valore che il Luogotenente i riceve dal generale. Il Luogotenente i agisce da Generale nell'Algoritmo OM(M-1) per mandare il valore v_i ad ognuno degli altri n-2 Luogotenenti.
3. Per ogni i, e ogni $j \neq i$, sia v_j il valore che il luogotenente i ha ricevuto dal luogotenente j nel passo 2 (usando l'algoritmo(M-1)).
   Il luogotenente i usa il valore *majority* ($v_1, v_2, \ldots v_n$).

L' algoritmo ha un caso base (per *M=0*), e viene invocato ricorsivamente per *M>0*. I passi ricorsivi si svolgono in due fasi, nella prima fase (punto 1 e 2) avvengono gli scambi di messaggi attraverso cicli continui di invio e ricezione. Nella seconda fase (punto 3) invece i processi utilizzano tutte le informazioni scambiate precedentemente per giungere a una conclusione. Parleremo di iterazione k dell'algoritmo per indicare la k-esima invocazione dell'algoritmo da parte di un processo. OM(M) è la prima iterazione dell'algoritmo (il generale invia il comando); OM(M-1) è la seconda iterazione

dell'algoritmo ed, in questo caso, l'iterazione viene eseguita da ogni Luogotenente (ogni Luogotenente invia il valore ricevuto dal comandante agli altri Luogotenenti).

Sebbene la sua definizione sia intuitiva, l' implementazione dell'algoritmo è complessa e richiede un numero elevato di scambi di messaggi. Inoltre il sistema deve essere sincrono. Si deve sapere quando tutti i processi hanno inviato i loro messaggi per considerare conclusa una fase dell'algoritmo.

Nel seguito mostriamo dettagliatamente le due fasi dell'algoritmo a partire da esempi.

3.3 Problematiche e analisi dettagliata del funzionamento dell'algoritmo

Descriveremo ora in dettaglio le due fasi dell' algoritmo definito precedentemente partendo dalle considerazioni in **[8]**, in modo da comprenderne il funzionamento e le possibili problematiche. A partire da questa implementazione, pensata per funzionare in centralizzato, adatteremo ed estenderemo l'implementazione all' ambiente distribuito quale è il caso di questo lavoro.

Essa fa uso di un algoritmo di decisione finale che sfrutta le potenzialità delle strutture dati utilizzate per l' immagazzinamento dei dati.

3.3.1 Fase 1 dell'algoritmo: lo scambio di messaggi

La prima fase dell' algoritmo è una raccolta dati che avviene attraverso le iterazioni di più cicli di scambio messaggi. Il numero di cicli da effettuare dipende dal numero di possibili processi Traditori M, ossia dal massimo numero di processi nel nostro sistema che si assume possano malfunzionare.

Dato M, saranno necessari M+1 cicli di scambio messaggi per essere certi di giungere a una soluzione concordata tra tutti i processi.

L'iterazione 0 è triviale: il Generale invia a tutti i suoi *N-1* Luogotenenti il proprio valore di proposta di decisione (il proprio ordine): il Generale si mette in stato di riposo, nessuno invia ulteriori messaggi ad esso, né lui invierà altri messaggi successivamente.

A questo punto, secondo la definizione, per ogni messaggio ricevuto, ogni Luogotenente agisce da Generale nell'algoritmo *O(M-1),* considerando tutti gli altri processi come propri Luogotenenti, rilanciando l'informazione ricevuta a tutti gli altri *N-2* processi.

Il significato di questi messaggi è avvisare gli altri Luogotenenti che "nell' iterazione i-esima, mi è stato detto dal processo j, che ha ricevuto dal processo dal processo j', la decisione di '*attaccare*' ".

Una volta ricevuti anche questi messaggi, inizia una nuova iterazione di rilancio dei messaggi ricevuti e cosi via: possiamo notare fin da ora, sebbene in seguito lo quantificheremo con precisione, come il numero di messaggi scambiati cresce drasticamente con *M*. Vediamo come implementare nella pratica tutto questo.

Nelle iterazioni successive alla 0, ogni processo Luogotenente, ricevuto l'ordine dal Generale, costruisce un insieme di messaggi: ogni messaggio è composto da una tupla che contiene un valore e un percorso. Il valore, nel caso di algoritmo di consenso binario che stiamo analizzando, è semplicemente "`attaccare`" o "`ritirata`", ossia **1** o **0**.

Il percorso invece è un insieme di identificatori di processo del tipo ID1, ID2,…..IDN, che serve per memorizzare da chi ha avuto origine quella particolare informazione e lungo quale percorso si è sviluppata.

In sostanza la tupla **{1, ID_1 ID_2 ….. ID_N }** indica che nell' iterazione N, il processo identificato da ID_N sta dicendo che nell' iterazione N-1 al processo ID_{N-1} è stato detto da ID_{N-2} che nell' iterazione N-2 è stato detto da ID_1 che il valore trasmesso dal Generale è 1.

Cominciamo quindi con analizzare un primo semplice esempio: supponiamo di avere un sistema con 7 Processi, un Processo Generale P0 e sei Processi Luogotenenti P1, P2, P3, P4, P5 e P6. Supponiamo di avere al più un processo gusto: M=1.

Supponiamo inoltre che il Generale comandante sia il traditore, e comunichi di proposito un valore di 0 ai primi 3 Luogotenenti, P1, P2 e P3, e il valore di 1 agli altri Luogotenenti rimanenti P4, P5, P6.

Verrà eseguito *OM(1).*

Durante l'esecuzione del Punto 1 di *OM(1),* semplicemente il Generale costruisce e invia due semplici tuple {0, 0} per i primi 3 Luogotenenti e {1,0} per i rimanenti.

Durante l'esecuzione del Punto 2 di *OM(1),* ogni Luogotenente agisce come Generale comandante in *OM(0).* Ogni processo ricevuto il messaggio, aggiunge il proprio ID al campo "percorso" della tupla se e solo se questo non è gia presente (non sono concessi percorsi ciclici nella tupla cioè) e reinvia questo messaggio a tutti gli altri (tranne che al Generale comandante).

Nel nostro esempio, verrebbero costruiti e spediti i messaggi illustrati in formato tabellare nel seguito. La tabella va letta colonna per colonna, dove troviamo il messaggio spedito dai singoli Luogotenenti con la relativa destinazione. La colonna "**Sender=P_i** " mostra i messaggi inviati in OM(0) dal Luogotenente i-esimo.

Sender=P_1		**Sender=P_2**		**Sender=P_3**		**Sender=P_4**		**Sender=P_5**		**Sender=P_6**	
Dest	Msg	Dest	Msg	Dest	Msg	Dest	Msg	Dest	Msg	Dest	Msg
-	-	P_1	{0,02}	P_1	{0,03}	P_1	{1,04}	P_1	{1,05}	P_1	{1,06}
P_2	{0,01}	-	-	P_2	{0,03}	P_2	{1,04}	P_2	{1,05}	P_2	{1,06}
P_3	{0,01}	P_3	{0,02}	-	-	P_3	{1,04}	P_3	{1,05}	P_3	{1,06}
P_4	{0,01}	P_4	{0,02}	P_4	{0,03}	-	-	P_4	{1,05}	P_4	{1,06}
P_5	{0,01}	P_5	{0,02}	P_5	{0,03}	P_5	{1,04}	-	-	P_5	{1,06}
P_6	{0,01}	P_6	{0,02}	P_6	{0,03}	P_6	{1,04}	P_6	{1,05}	-	-

Durante l'esecuzione del punto 1 di OM(1), abbiamo invece di 7 messaggi, uno per ogni Luogotenente. Nella fase successiva, il numero di messaggi scambiati aumenta perche', ogni Luogotenente dovra' inviare 5 messaggi (uno ad ogni Luogotenete eccetto lui stesso). Infatti durante l'esecuzione di OM(0), il numero di messaggi scambiati da ogni Luogotenete è uguale a 5*6.

Al solito, ogni processo prima dell'invio dei messaggi, aggiunge il proprio identificatore ID al campo "percorso" della tupla.

Se il numero di traditori aumenta, aumentano le iterazioni dell'algoritmo OM. Nel caso di 2 traditori i messaggi inviati all' iterazione successiva dell'algoritmo sono i seguenti (in cui per comodità adesso omettiamo la destinazione visto che vengono ad ogni modo inviati in broadcast a tutti i processi):

Sender=P_1	**Sender=P_2**	**Sender=P_3**	**Sender=P_4**	**Sender=P_5**	**Sender=P_6**
{0,021} {0,031} {1,041} {1,051} {1,061}	{0,012} {0,032} {1,042} {1,052} {1,062}	{0,013} {0,023} {1,043} {1,053} {1,063}	{0,014} {0,024} {0,034} {1,054} {1,064}	{0,015} {0,025} {0,035} {1,045} {1,065}	{0,016} {0,026} {1,036} {1,046} {1,056}

In questa tabella, ad esempio, l' ultimo messaggio ***{1,056}*** ha il significato "P6 sta dicendo che all' iterazione precedente P5 gli ha detto che all' iterazione recedente il Generale comandante P0 gli ha ordinato **1**. Questo messaggio è inviato in broadcast a tutti gli altri processi.

Notiamo quindi che il numero di messaggi scambiati durante l' esecuzione dell' algoritmo è *(N-1)* all' iterazione 0, *(N-1)*(N-2)* all' iterazione 1, *(N-1)*(N-2)*(N-3)* all' iterazione numero 2, e cosi via fino all ultima iterazione.

Poiché il numero di iterazioni da svolgere dipende da M, cioè dal numero di processi Traditori sul totale N, abbiamo che il numero di messaggi in totale scambiati durante l' esecuzione di *OM(M)* è pari a:

$$\textit{Num. messaggi totale} = \frac{N!}{(N-M)!}$$

Questo può portare notevoli problemi di traffico in sistemi dove *N* e *M* sono grandi, la scalabilita' e' il principale problema di questo algoritmo.

3.3.2 Fase 2 dell'algoritmo: la fase di decisione

L'iterazione dell'algoritmo che termina per prima è quella eseguita per ultima. Al termine dello scambio di messaggi eseguito durante l'iterazione, inizia la fase di decisione del valore da utilizzare dal processo come valore candidato alla decisione finale nell'iterazione precedente. La fase di decisione è di fondamentale importanza per consentire ai generali leali di accordarsi sullo stesso valore, e quindi raggiungere il consenso. Nella fase 2 dell'algoritmo ogni luogotenente si comporta come segue:

Per ogni i, e ogni j ≠ i, sia v_j il valore che il luogotenente i ha ricevuto dal luogotenente j nel passo 2 (usando l'algoritmo(M-1)). Il luogotenente i usa il valore *majority* (v_1, v_2, … v_n).

Questa fase dell' algoritmo presuppone quindi che precedentemente i processi abbiano conservato tutti i messaggi scambiati in una qualche struttura dati, a cui possano accedere adesso che è il momento di decidere.

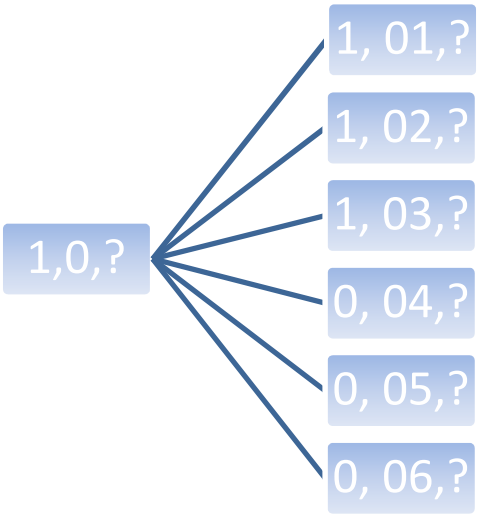
Per far questo, in **[8]** (una delle prime implementazioni in C dell' algoritmo, sebbene non in campo distribuito), è stata utilizzata una struttura dati ad albero organizzata come segue.

Ogni iterazione corrisponde ad un livello dell' albero, ogni nodo dell' albero è invece composto da 3 campi: un valore di ingresso, uno di uscita e un percorso. Il valore di ingresso è semplicemente il valore propagato di volta in volta nei messaggi, il valore di uscita viene lasciato indeterminato fino alla fase di decisione, e il percorso indica la sequenza di nodi da cui è stato propagato il messaggio.

Come esempio consideriamo la situazione già illustrata di 7 processi di cui uno Traditore, e supponiamo che il traditore sia il Generale comandante che invia 1 ("attacco") a P1, P2 e P3, mentre invia 0 ("ritirata") a P4, P5, P6. Per raggiungere il consenso verrà eseguito OM(1).

L'invocazione OM(1) comporterà l'invocazione di OM(0) da parte di ogni Luogotenente. Seguita comporterà la chiamata di OM(0) da parte di ogni Luogotenente. Quando termina OM(0) su ogni processo, seguirà la fase di decisione di OM(1). Avremmo due sole iterazioni, per cui avremmo un albero a due livelli: la radice e suoi figli.

L' albero risultante su ogni processo, che chiameremo *Information Tree*, sarà del tipo seguente. L'albero mostrato di seguito corrisponde a quello costruito sul processo P1:



Al primo livello troviamo l'informazione inviata dal Generale comandante a P1. Nel nostro esempio era 1 ("attacco").

Al secondo livello troviamo i messaggi ricevuti da P1 ed inviati dagli altri Luogotenenti . Per esempio, il Luogotenete P2 (che è leale) comunica a P1 che lui ha ricevuto 1 ("attacco") da P0. Lo stesso farà P3. Mentre P4, P5 e P6, comunicheranno che hanno ricevuto 0 ("ritirata"). In questo semplice caso, poiché P0 è il solo processo faulty, il secondo livello dell' Information Tree è identico per tutti i processi.

Ricordiamo che a questo punto tutti i valori di uscita di ogni nodo sono ancora sconosciuti e indefiniti pari a "?".

Una volta che sono terminate le operazioni di invio messaggi, l'albero è stato completato ed iniziano le fasi di decisione delle varie iterazioni dell'algoritmo. Ogni processo deciderà in locale in base alle informazioni disponibili, quale sarà il valore da usare nella fase di decisione successiva. Per far questo viene utilizzata la funzione **`majority()`** vista in precedenza che calcola per ogni livello dell'albero i valori di uscita e li assegna ai nodi dell'albero del livello precedente, risalendo la struttura fino alla radice. Nel dettaglio, l'implementazione dell' algoritmo di decisione è realizzata come segue:

1. Ogni foglia dell'albero copia il valore di ingresso in quello di uscita

2. A partire dal grado dell' albero pari a *M-1* e fino al grado 0, il valore di uscita di ogni nodo padre interno viene calcolato sulla base della funzione **majority()** tra tutti i valori di uscita di tutti i suoi figli. In caso di parità è presente un valore di default unico per tutti i processi (**default**, nel seguito) che deve essere utilizzato.

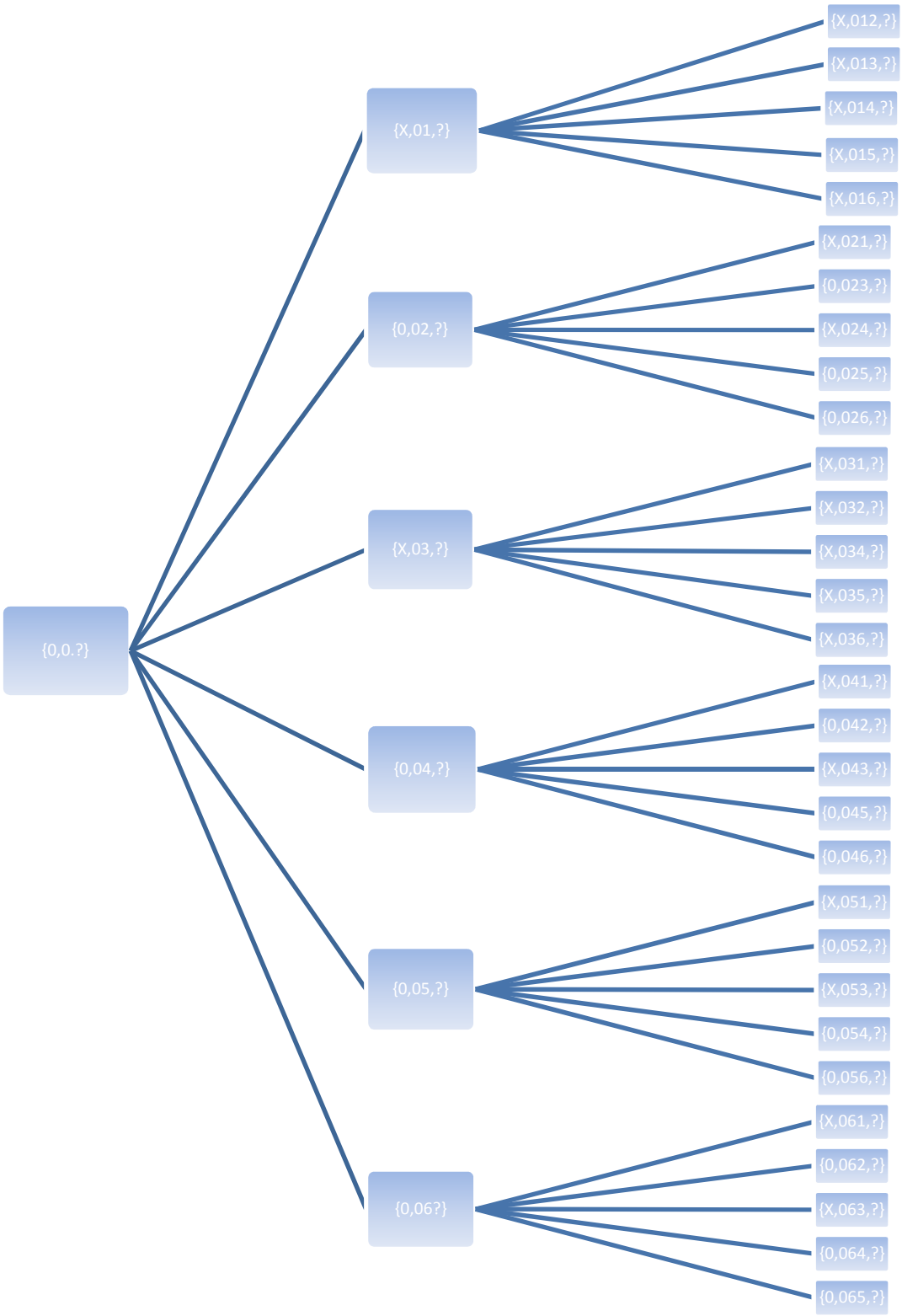
3. Il valore di uscita del nodo radice, è il risultato finale dell'algoritmo e rappresenta il comando su cui viene raggiunto il consenso.

Consideriamo lo scenario illustrato in figura precedente: al primo passo, ciascun nodo foglia semplicemente copia il valore di ingresso in quello di uscita. Per determinare quanto vale il valore di uscita del nodo padre, invochiamo la funzione `majority(1, 1, 1, 0, 0, 0)` che restituisce il valore di default. Il valore di uscita del nodo padre viene posto pertanto pari a **default**. In questo semplice scenario, con un *Information Tree* a due soli livelli, l' algoritmo di Lamport termina, restituendo **default**. Tutti i processi sceglieranno 1 e si accorderanno dunque su quel comando.

Per comprendere appieno il funzionamento dell' algoritmo passiamo ad analizzare il caso che prevede due processi guasti. In questo scenario avremo un Information Tree a 3 livelli.

Assumiamo nel seguito di avere 7 processi, di cui 2 faulty, ad esempio il processo 1 e il processo 3, e che il Generale comandante invii il comando 0 ("ritirata").

In questo caso M=2 e per raggiungere il consenso verrà invocato OM(2). Sono necessarie 3 iterazioni dell'algoritmo. Lo schema dell'Information Tree su ogni processo è mostrato nella pagina seguente. Assumiamo che P1 e P3 siano i processi faulty.


{0,0.?}
{X,01,?}
{0,02,?}
{X,03,?}
{0,04,?}
{0,05,?}
{0,06?}
{X,012,?}
{X,013,?}
{X,014,?}
{X,015,?}
{X,016,?}
{X,021,?}
{0,023,?}
{X,024,?}
{0,025,?}
{0,026,?}
{X,031,?}
{X,032,?}
{X,034,?}
{X,035,?}
{X,036,?}
{X,041,?}
{0,042,?}
{X,043,?}
{0,045,?}
{0,046,?}
{X,051,?}
{0,052,?}
{X,053,?}
{0,054,?}
{0,056,?}
{X,061,?}
{0,062,?}
{X,063,?}
{0,064,?}
{0,065,?}

Per lavorare in maniera indipendente dal comportamento dei processi faulty abbiamo indicato appositamente con **x** i valori che un processo faulty può decidere di inviare per fuorviare gli altri processi. L'algoritmo deve consentire di raggiungere il consenso, indipendentemente dai valori 0 o 1 di **x**. Inoltre Il valore di x usato in P1 è indipendente dal valore di **x** usato in P3.

E' possibile notare dalla struttura dell' albero come i valori errati **x** dei processi faulty, man mano che scendiamo lungo la profondità dell'albero aumentino considerevolmente: questo perché questi valori cominciano a provenire non solo da processi faulty, ma anche da quei processi corretti che includono valori che sono passati attraverso processi faulty nel loro percorso.

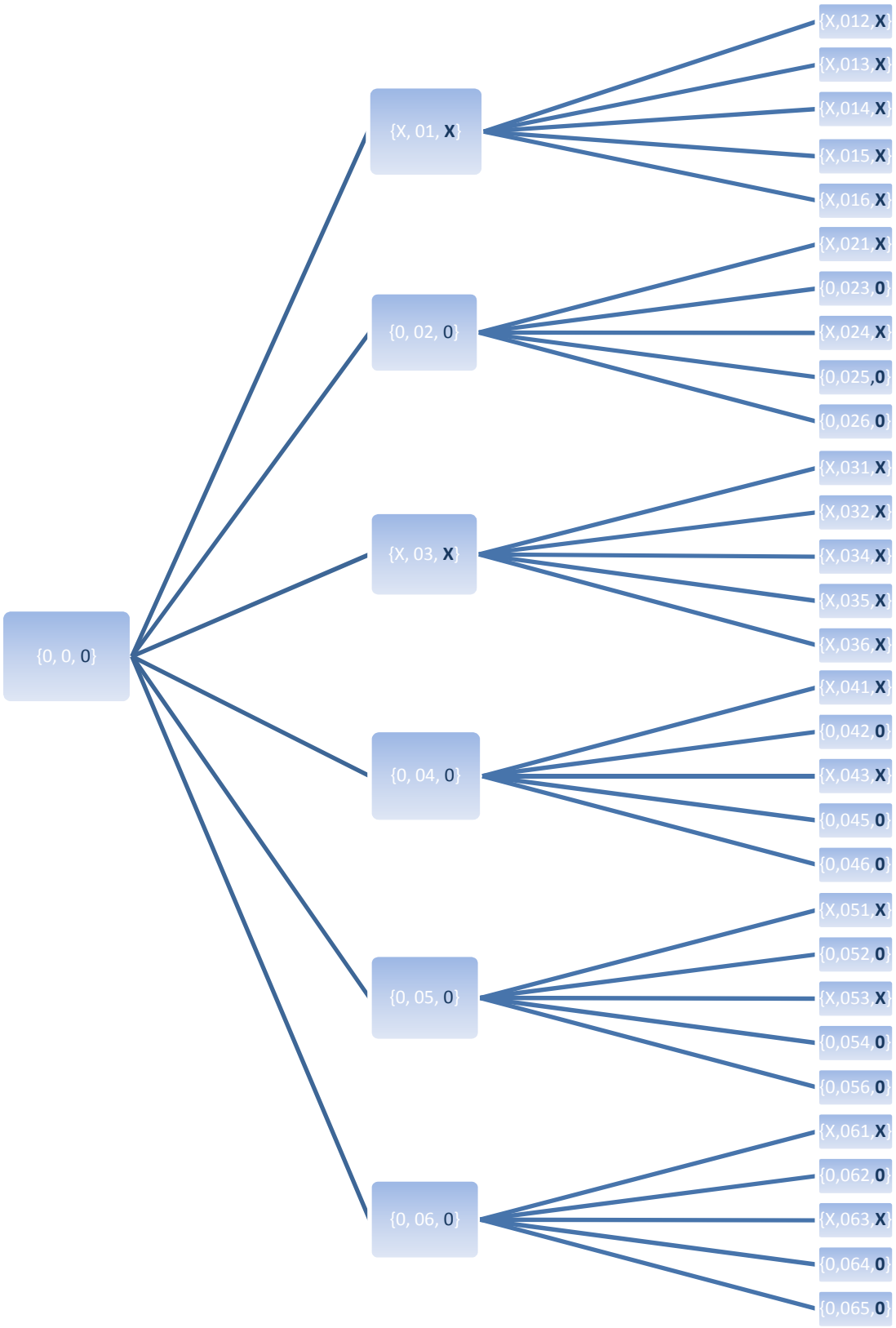
Come risultato di questo comportamento, sui nodi foglia abbiamo ben 18 valori faulty su 30. E' chiaro fin da subito che eseguendo semplicemente una scelta a maggioranza su questi valori e poi terminando non è possibile raggiungere il consenso.

Ogni processo ricava il valore corretto di decisione sfruttando la struttura dell'albero.

Ad ogni nodo foglia viene quindi assegnato sul valore di output quello di input ottenendo la struttura che riportiamo.

Nel grafico della pagina seguente sono stati calcolati i valori di output anche dei nodi intermedi padre (in rosso), sulla base della funzione **majority()** calcolata sui valori di output dei loro nodi figli (le foglie in questo esempio), arrivando fino alla radice, in cui è evidenziato il valore di ingresso che corrisponde al valore di decisione finale dell' algoritmo.

Nella pagina che segue riportiamo l'Information Tree al termine dell'algoritmo per un processo Pi.


{0, 0, 0}
{X, 01, X}
{0, 02, 0}
{X, 03, X}
{0, 04, 0}
{0, 05, 0}
{0, 06, 0}
{X,012,X}
{X,013,X}
{X,014,X}
{X,015,X}
{X,016,X}
{X,021,X}
{0,023,0}
{X,024,X}
{0,025,0}
{0,026,0}
{X,031,X}
{X,032,X}
{X,034,X}
{X,035,X}
{X,036,X}
{X,041,X}
{0,042,0}
{X,043,X}
{0,045,0}
{0,046,0}
{X,051,X}
{0,052,0}
{X,053,X}
{0,054,0}
{0,056,0}
{X,061,X}
{0,062,0}
{X,063,X}
{0,064,0}
{0,065,0}

Come si nota i percorsi {01} e {03} , quelli che comprendono i processi faulty dello scenario, P1 e P3, hanno una evidente maggioranza di valori faulty **x**, per cui i valori di output dei loro nodi intermedi vengono settati a **x**.

La particolare struttura dell' albero e dell' algoritmo di decisione permette ancora di giungere al consenso, visto che sul livello 1, i valori **x** sono in minoranza rispetto agli altri: il successivo roll-up dell' algoritmo verso la radice, porta ad assegnare 0 al valore di output della radice, che è poi proprio l'ordine originario impartito dal Generale P0.

L'algoritmo di agreement binario è alla base di molti altri algoritmi di consenso. Come esempio, descriveremo nel seguito la risoluzione dell'algoritmo di agreement multivalued attraverso l'algoritmo di agreement binario.

3.4 Estensione dell' algoritmo al caso multivalued

Illustriamo ora l' estensione teorica al caso di consenso multivalued così come presentata in **[6]**. Per problema del consenso multivalued si intende il problema di accordarsi su valori che fanno parte di un dato dominio *D* qualunque, non solo appartenenti al dominio [0,1].

Anche per questo algoritmo, il consenso è raggiunto sotto la condizione che su N processi non possono esserci più di M processi faulty , con N >= 3M+1.

Ciascun processo del sistema mantiene in memoria le seguenti 3 variabili locali: due array di booleani, rispettivamente **`v[]`** e **`p[]`**, e un booleano **`alert`**.

L'algoritmo funziona nel modo seguente:

1. Nel primo round ciascun processo spedisce il suo valore iniziale a tutti gli altri processi. I processi memorizzano i valori ricevuti insieme a quello proposto da loro stessi all'interno dell'array **v[]**. A questo punto il processo analizza il contenuto dell'array per vedere se egli è un processo ***"perplesso".***
    a. Un processo viene definito **perplesso** se riceve almeno (N-M)/2 valori iniziali proposti dagli altri, diversi dal proprio. Processi che non sono **perplessi** vengono invece definiti ***"contenti"***.
    b. A seconda o meno che il processo sia perplesso, il flag booleano p[i] viene settato a **true** o a **false** rispettivamente.

2. Nel secondo round ciascun processo **perplesso** *i*, spedisce un messaggio a tutti gli altri processi del sistema per informare che *i* è un processo **perplesso**. Il significato del messaggio contenente **p[i]** è semplicemente *"Io sono perplesso"*.

3. Una volta ricevuti tutti i messaggi del secondo round, il booleano **alert** viene settato a **true** se e solo se almeno [N-2M] elementi dell' array **p[]** sono pari a 1.

Una volta compiuti questi 3 passi, inizia la computazione dell' **algoritmo di agreement binario** per accordarsi sul valore binario di **`alert`.**

Al termine dello stesso, i casi possibili sono due:

A. I processi si accordano sul valore **true**: in questo caso significa che ci sono processi corretti con valori differenti del dominio V. Tutti i processi corretti usano pertanto un valore di default predefinito come risultato dell' algoritmo di agreement multivalued.
B. I processi si accordano sul valore false: in questo caso significa che tutti i processi corretti e **contenti** concordano sullo stesso valore iniziale del dominio V. Usano pertanto questo valore

come risultato dell' **algoritmo di agreement multivalued**. I processi **perplessi** deducono lo stesso identico valore di V trovato dai processi **contenti**, semplicemente utilizzando il valore iniziale comune alla maggioranza dei processi *"contenti"*. In altri termini, prendono in esame tutti i valori **v[j]** per i quali il corrispondente **p[j]** è **false.**

Il valore tra questi con l'occorrenza maggiore, è il risultato della computazione dell' **algoritmo di agreement multivalued** per i processi **perplessi.**

CAPITOLO 4

AMBIENTE OMNeT++

Non è ovviamente nostra intenzione, in questa sede, appronfondire fin nei dettagli le potenzialità e le pecurialità di OMNeT++, e il funzionamento dei suoi numerosi tool.

Vogliamo semplicementare dare una presentazione di questo ambiente di sviluppo e di simulazione, il perché è stato scelto per il nostro lavoro, quali sono i concetti che ne stanno alla base del funzionamento e gli strumenti che mette a disposizione del programmtore, e, infine, dare una panoramica su come avviene lo sviluppo di un modello OMNeT++.

Per tutti i dettagli e gli approfondimenti qualora fosse necessario, rimandiamo al Manuale di OMNeT++ e alla documentazione disponibile in rete **[10]**.

4.1 Perché OMNeT++?

OMNeT++ è una libreria di simulazione, estensibile, modulare e basata su linguaggio C++. La libreria inoltre implementa e mette a disposizione del programmatore una piattaforma di simulazione, corredata da un ambiente grafico runtime.

OMNeT++:

- E' facilmente estensibile e adattabile alle diverse applicazioni che il programmatore intende sviluppare: una volta esteso con i sorgenti delle nostre applicazioni basta ricompilarne il codice. Ne esistono inoltre diverse estensioni: per la simulazione real-time, per la simulazione di reti, per linguaggi di programmazione (C++, Java, etc), per database integration, etc.

- E' facilmente portabile e supportato da diverse piattaforme e sistemi operativi: è eseguibile su Linux, Mac OS X, altri sistemi Unix-like e sui sistemi Windows (XP, Win2K, Vista, 7).

- OMNeT++ è inoltre del tutto *free* per scopi accademici e di ricerca, e sta diventando rapidamente una delle piattaforme di simulazione più diffuse nella comunità di ricerca scientifica.

  E' possibile scaricarne i sorgenti e tutta la documentazione necessaria in rete, all' URL http://www.omnetpp.org.

  Per scopi commerciali e di profitto, è possibile utilizzare la versione OMNEST, la quale però richiede il pagamento delle licenze.

4.2 Introduzione a OMNeT++

4.2.1 Contenuto della distribuzione

Una volta scaricati e installati i sorgenti della distribuzione, il contenuto della main directory di OMNeT++ appare come segue:

omnetpp/ OMNeT++ root directory

- **bin/** eseguibili OMNeT++
- **include/** file di intestazione per modelli della simulazione
- **lib/** file di libreria
- **bitmaps/** file immagini che possono essere usati per la grafica della simulazione
- **doc/** manuale (PDF), readme, licenze, etc.
  - **manual/** manuale in HTML
  - **tictoc-tutorial/** tutorial di introduzione a OMNeT++
  - **api/** API in HTML
  - **nedxml-api/** API reference per la librera NEDXML
  - **src/** sorgenti della documentazione
- **src/** sorgenti OMNeT++
  - **nedc/** nedtool, compiler dei messaggi
  - **sim/** kernel di simulazione
  - **parsim/** file per l' esecuzione distribuita
  - **netbuilder/** sorgenti per la lettura dinamica dei file NED
  - **envir/** codice per le interfacce utente
  - **cmdenv/** interfaccia utente a riga di comando
  - **tkenv/** tk: interfaccia utente grafica
  - **gned/** editor NED grafico
  - **plove/** plotting tool per output di tipo vettoriale
  - **scalars** plotting tool, output di tipo scalare

nedxml/ libreria NEDXML

utils/ varie utility

test/ contiene gli strumenti per vari test (es. test di regressione)

core/

distrib/

Nella directory samples/ sono contenuti alcuni esempi di simulazioni per un primo approccio con OMNeT++.

E' possibile trovare directory addizionali come ad esempio **msvc/** (che contiene i componenti necessari per il Microsoft Visual C++) nel caso di installazione su sistemi Windows.

4.2.2 Cos'è OMNeT++

Più nel dettaglio OMNeT++ è un ambiente di simulazione a **eventi discreti**. La sua area di applicazione primaria riguarda la modellazione e lo studio di protocolli di reti, ma proprio grazie alla sua architettura flessibile e generica, oltreché facilmente espandibile, è possibile utilizzarla con successo anche in molti settori quali ad esempio:

- Modellizzazione del traffico di reti di telecomunicazione
- Modellizzazione di multiprocessori e altri sistemi hardware distribuiti
- Valutazione di architetture hardware
- Modellizzazione e valutazione degli aspetti di performance di sistemi complessi
- Modellizzazione di qualsiasi altro sistema dove è opportuno un approccio a eventi discreti.

4.2.3 Simulazione ad Eventi Discreti (DES)

Un sistema a eventi discreti è un sistema in cui gli eventi che lo caratterizzano accadono a istanti discreti nel tempo, e impiegano tempo zero per completarsi (sono istantanei).

Si assume quindi che tra due eventi consecutivi nel tempo, non accada nulla: in altre parole, tra due eventi consecutivi, lo stato del sistema resta invariato.

Tipicamente le reti di computer possono essere modellate come sistemi a eventi discreti: gli eventi ad esempio possono essere l'inizio di una trasmissione di un pacchetto, la sua ricezione, lo scadere di un timeout etc.

I sistemi in cui il modello a eventi discreti è un'assunzione valida, possono essere modellati con la Simulazione a Eventi Discreti, propria di OMNeT++.

All'interno di queste simulazioni ci si riferisce al *simulation time* come al tempo interno al modello in cui accadono gli eventi, in contrapposizione al *real time (o CPU-time)*, che si riferisce al tempo trascorso da quando la simulazione è stata avviata.

4.2.4 Funzionamento delle simulazioni in ambiente OMNeT++

OMNeT++ mantiene l'insieme degli eventi futuri in una struttura dati denominata FES (Future Event Set) e funziona seguendo il seguente pseudo-codice:

```
1. inizializzazione  //costruzione del modello e inserimento
                     //degli eventi iniziali nella struttura FES
```

```
2. while (FES non vuota e la simulazione non ancora completata)
   {
        recupera il primo evento dalla struttura FES
        t:= timestamp dell' evento
        processa evento  //questo può comportare l'inserimento di
                         //eventi nuovi nella struttura dati FES
                         //oppure l' eliminazione di eventi già
                         //esistenti
        }
3. termina simulazione  //output di statistiche, risultati etc.
```

- Inizialmente vengono costruite le strutture dati che rappresentano il modello, in seguito viene richiamato il codice di inizializzazione definito dal programmatore, e infine inseriti nella FES gli eventi iniziali per assicurare che la simulazione possa avviarsi con successo.
- Il `while` successivo semplicemente prende gli eventi contenuti nella struttura FES e li processa uno a uno **in rigoroso ordine di tempo** (ad ogni evento è associato un timestamp) per mantenere la **causalità del sistema**.

  Processare un evento include l'esecuzione di codice definito dal programmatore, e può succedere quindi che nuovi eventi vengano inseriti all' interno della FES, oppure che venga cancellato qualche evento già presente (esempio: cancellazione di un timeout non più utile).
- La simulazione semplicemente termina quando non ci sono più eventi da schedulare oppure quando non c'è più bisogno di procedere perché il *simulation time* (o anche il *CPU-time*) ha raggiunto un limite predefinito dall' utente.

4.3 Architettura e Modello di OMNeT++

4.3.1 Componenti di un Modello OMNeT++

Un modello OMNeT++ è costituito dai seguenti componenti:

- *Un insieme di file con estensione .NED* scritti in linguaggio NED (**NE**twork **D**escription) che descrivono la topologia e la struttura della rete, i suoi moduli, come questi sono connessi, i parametri etc. I file .NED possono essere scritti usando un qualsiasi editor di testo disponibile.
- *Un insieme di file con estensione .msg* per la definizione dei messaggi (pacchetti) che verranno scambiati tra i moduli che compongono il modello. E' possibile definire vari tipi di messaggi composti da più campi di dati. OMNeT++ in fase di compilazione tradurrà queste definizioni di messaggi in classi C++ opportunamente strutturate e dotate di metodi per l'accesso ai campi dati (metodi di tipo get e set, per leggere scrivere valori nei campi dati del pacchetto).
- *File sorgenti dei Simple Modules del modello*. Questi sono file sorgenti C++ con estensione .h/.cc
- *Il Kernel di Simulazione*. Contiene il codice, scritto in C++, che gestisce la simulazione e alcune librerie di classe che vengono compilate con esso.

Le applicazioni di simulazione vengono costruite a partire dai componenti sopraenunciati.

Prima di tutto i file .msg vengono tradotti in file in codice sorgente C++ tramite il programma opp_msgc. A questo punto tutti i file sorgenti C++ vengono compilati assieme al kernel per ottenere l'eseguibile della simulazione.

Una volta che l'eseguibile viene lanciato, per prima cosa legge il file di configurazione *omnettpp.ini* che può contenere varie informazioni sul modello costruito: parametri di ingresso, simulation runs per vari scenari possibili e altro.

Cominceremo a dare uno sguardo a ciascuno di essi.

4.3.2 Topologia della rete: Simple Modules e Compound Modules

L'architettura di un modello OMNeT++ consiste di più *moduli* annidati in modo gerarchico: i vari moduli vengono via via assemblati per costruire *component* più grandi attraverso l' uso di un linguaggio ad alto livello, il NED.

Il building block su cui si basa l'architettura OMNeT++ viene definito ***Simple Module***: questi sono i moduli di livello più basso nella scala gerarchica, sono programmati e scritti in linguaggio C++ utilizzando la libreria di simulazione, e ne implementano il comportamento. Il modulo top-level è invece il **Modulo di Sistema** (*System Module*).

La profondità della gerarchia non è limitata in nessun modo: il System Module contiene alcuni *Submodules,* i quali a loro volta contengono due o più *Submodules,* e cosi via. I moduli che contengono Submodules sono definiti *Compound Modules.*

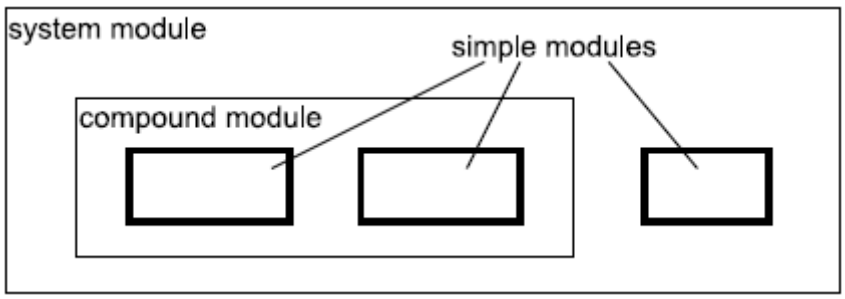


Figura 4.1: Simple e compound modules

4.3.3 Tipi di Modulo (Module Types)

Sia i moduli semplici che i moduli composti sono istanze di **module types**. Mentre descrive il modello, l'utente definisce i propri tipi di modulo: istanze di questi tipi servono come componenti per moduli più complessi.

L'utente crea il System Module come un'istanza di un tipo di modulo precedentemente definito: tutti i moduli della rete sono istanziati come sottomoduli e sotto-sottomoduli di un modulo di sistema.

Una volta che un tipo di modulo è usato come *building block*, non ci sono più distinzioni tra modulo complesso o semplice. Questo permette all'utente di dividere un modulo semplice in più moduli semplici inclusi in un modulo complesso, o viceversa, aggregare la funzionalità di un modulo complesso in un singolo modulo semplice, senza che gli eventuali utilizzatori del modulo ne vengano influenzati.

Il tipo di modulo può essere memorizzato in file separatamente dal luogo del loro attuale utilizzo. E' quindi possibile per l'utente raggruppare tipi di modulo esistenti per creare *Component Libraries*

.

4.3.4 Moduli e Parametri

Ogni Modulo può essere parametrizzato: i parametri possono essere assegnati direttamente nel NED File che descrive il Simple Module in questione, oppure essere inseriti nel **file di configurazione `Omnetpp.ini`.**

I parametri possono essere usati per customizzare il comportamento di un determinato Simple Module oppure per parametrizzare la topologia della rete.

All'interno di un Compound Module i parametri possono definire il numero di sottomoduli che lo compongono, il numero di gates di ciascuno Simple Module o il modo in cui costruire le connessioni interne.

I parametri di un modulo possono prendere in ingresso valori booleani, stringhe e valori numerici, questi ultimi a loro volta possono includere *espressioni*, chiamate di funzioni C, variabili random o essere immesse interattivamente dall'utente, al lancio della simulazione.

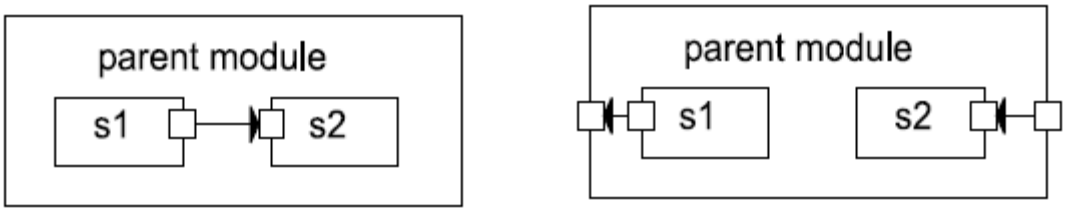
4.3.5 Messaggi, Gates, Links

I moduli comunicano tramite scambio di messaggi. In una simulazione, i messaggi possono rappresentare frame o pacchetti in una rete di computer, jobs o costumers in una *queuing network* o altri tipi di entità mobili.

E' possibile inviare messaggi sia direttamente alla loro destinazione o lungo un percorso definito da come il programmatore struttura la rete, attraverso una serie gates e connessioni. E' possibile inoltre per un modulo, spedire messaggi anche a sé stesso, funzionalità solitamente utilizzata per l' implementazione di *timers* e *trigger* vari. **Il *local simulation time* di un modulo avanza ogniqualvolta questo riceve un messaggio.**

Ogni modulo, per lo scambio di messaggi, è dotato opportunamente di interfacce di ingresso/uscita definite ***gates***: i messaggi sono spediti fuori attraverso gates di uscita e attivano attraverso *gates* di entrata: i gates dei vari moduli sono poi connessi tra loro tramite i ***link***.

Ogni connessione (link) è creata all' interno di un singolo livello della gerarchia dei moduli; dentro un modulo complesso si possono connettere i corrispondenti moduli di due sottomoduli, oppure un gate di un sottomodulo e un gate di un modulo complesso, così come illustrato in figura:



a)Submodules connessi tra loro *b)Submodules connessi con il padre*

Figura 4.2: Connessioni di submodules

I messaggi viaggiano quindi attraverso la struttura gerarchica della rete da Simple Modules verso Simple Module. I Compound Module si comportano in modo **trasparente, rilanciando semplicemente i messaggi ricevuti**. Le varie connessioni che portano da un Simple module ad un altro Simple Module prendono il nome di *routes.*

4.3.6 Trasmissione dei Pacchetti

Per modellare la nostra rete in modo più facile, a ciascuna connessione è possibile assegnare 3 parametri per caratterizzarla:

- *propagation delay:* il tempo che impiega un pacchetto ad arrivare a destinazione.
- *bit error rate:* la probabilità che un bit di informazione sia trasmesso non correttamente, impiegato per modellare canali con rumore.
- *data rate:* in bit/s, impiegata per il calcolo del transmission time di un pacchetto.

Per il programmatore è possibile assegnare uno o più di questi parametri per ogni singolo link individualmente, oppure definire *link types* e utilizzare questi per un insieme di connessioni. Ognuno di questi parametri è comunque opzionale.

Secondo il modello OMNeT++ inoltre, l' invio di un messaggio corrisponde all' invio del primo bit dello stesso, e la ricezione di un messaggio corrisponde alla ricezione dell' ultimo bit dello stesso. **Questo tipo di modello non è ovviamente sempre applicabile:** un esempio banale potrebbe essere la trasmissione di *flussi di bit,* come accade nei protocolli FDDI e con Token Ring, dove non si aspetta la ricezione completa di un pacchetto prima di rilanciarlo ma si trasmette subito il primo bit dello stesso non appena viene ricevuto.

4.3.7 Descrizione della Topologia della rete

La descrizione completa della topologia della rete viene effettuata dal programmatore secondo le proprie esigenze tramite il linguaggio NED messo a disposizione dal simulatore. Si tratta di un linguaggio dalla sintassi abbastanza semplice e intuitiva e ben documentato da tutorial e manuali, entrambi disponibili in rete. Non ci dilungheremo ovviamente in questa sede sulla descrizione della sua sintassi e della sua semantica, ci limitiamo semplicemente a segnalarlo, mentre per qualsiasi approfondimento rimandiamo a tutto il materiale facilmente reperibile e disponibile all' URL http://www.omnetpp.org/documentation.

CAPITOLO 5

LO STACK DI PROTOCOLLI RAPTOR

A partire dalla tecnologia ADS-B, il modulo SGT implementa la logica che si occupa della detection e della risoluzione di eventuali conflitti nelle informazioni scambiate dagli aerei. Il modulo SGT pone una notevole limitazione, assumendo in sostanza che il link di comunicazioni tra i vari soggetti coinvolti sia affidabile, cosa non vera in un ambiente a comunicazione wireless.

Lo stack di protocolli RAPTOR serve proprio a far da ponte tra l'SGT che impone questa assunzione, e l'ambiente *airborne self–separation* che non la soddisfa, operando in un sistema di comunicazione wireless potenzialmente inaffidabile.

In breve**, i protocolli RAPTOR sono algoritmi distribuiti** eseguiti da un numero di processi (gli aircraft) che si scambiano messaggi durante l'esecuzione da parte di ciascun processo della propria istanza dell'algoritmo. Ogni istanza dell'algoritmo, a fine esecuzione, produce lo stesso valore di uscita per tutti gli aircraft, in modo che i processi possano raggiungere varie forme di agreement.

Caratteristica fondamentale dei protocolli RAPTOR è che devono essere affidabili anche in presenza di guasti.

5.1 Modellazione del sistema

L'ambiente airborne self-separation viene modellato attraverso un set di processi (gli aircraft nel nostro caso) che scambiano informazioni attraverso un link wireless.

Il sistema viene quindi composto da un insieme statico di n processi $\prod = \{p_1, p_2, \ldots\ldots p_n\}$ che scambiano messaggi attraverso due primitive di comunicazioni molto semplici: `broadcast` e `deliver`, cosi definite. La primitiva `broadcast(t, ∏, m)` trasmette all'istante t, il messaggio m, a tutti i processi dell' insieme ∏, la primitiva `deliver(t, ∏)` ritorna l'insieme di messaggi trasmessi tramite l'invocazione di una `broadcast` dai processi in ∏ all' istante t.

Il sistema è inoltre sincrono: questo significa che i ritardi di comunicazione sono limitati da costanti note, come lo sono pure i tempi di calcolo di ciascun processo.
La comunicazione stessa tra processi avviene in modo sincronizzato. A ciascun istante t , ogni processo p_i appartenente a **∏** esegue le seguenti azioni:

1. invoca la broadcast(t, ∏, m),
2. invoca la deliver (t, ∏),
3. Esegue una transizione di stato che dipende sia dallo stato attuale che dal set di messaggi ricevuti all' istante t.

I processi sono modellati e implementati per funzionare in modo corretto, non esibiscono mai, cioè, un comportamento errato. Eseguono correttamente il protocollo definito sino al termine della sua esecuzione. Nella modellazione il fault di processo e il fault di comunicazione vengono modellati entrambi come un *faulty message transmission*: questo perché un crash all'interno di un processo, o l'esecuzione stessa di un processo scorretto, si riflettono sempre o nella perdita di messaggi, o nella loro corruzione.

La trasmissione di un messaggio da una sorgente p_i e una destinazione p_j viene definita *faulty* se accade una delle seguenti condizioni:

1. Omissione: il messaggio spedito da p_i non viene ricevuto da p_j

2. Addizione: viene consegnato un messaggio a p_j quando nessun messaggio era stato spedito da p_i.

3. Corruzione: il messaggio spedito da p_i, viene ricevuto da p_j in modo alterato.

I primi due punti comprendono ad esempio, il caso di un processo p_i il cui sistema di comunicazione wireless è mal funzionante, la terza invece il caso di un processo scorretto *(*o *malicious)* che invia messaggi dal contenuto diverso ai vari processi dell'insieme $\prod$. Una broadcast da parte di una sorgente p_i da luogo a n trasmissioni, una per ogni processo dell'insieme $\prod$, e ognuna di queste trasmissioni può risultare faulty. Il sistema è modellato per poter esibire e tollerare qualsiasi numero di *transmission fault* con la restrizione che questi, ad un istante t, non devono interessare più di f processi sorgenti con $n=3f+1$.

Si dice che un *transmission fault* interessa una sorgente p_i se interessa uno dei messaggi spediti da p_i all' istante t.

5.2 Il problema del consenso in RAPTOR

5.2.1 Il consenso binario

Il **consenso binario** (binary consensus) è il protocollo fondamentale dello stack RAPTOR, rappresenta l'agreement di base, ed è utilizzato come building block per forme di agreement più complesse implementate dagli altri protocolli dello stack.

Il consenso binario, come visto precedentemente, consente ai processi coinvolti nel nostro sistema di raggiungere un agreement su un valore tipo binario: ogni processo propone il proprio valore xi e alla fine dell'esecuzione dell' algoritmo tutti i processi si accordano sullo stesso valore d ∈ {0,1}.

Formalmente, le proprietà del protocollo sono definite dalle seguenti:

- **Validity**: Se tutti i processi propongono lo stesso valore v, allora è garantito che ogni processo che decide, decide v.

- **Agreement**: Qualsiasi coppia di processi non guasti si prenda in considerazione, questi non si accordano mai su valori differenti.

- **Termination**: Tutti i processi decidono entro *r* round con una probabilità P che tende a 1 al crescere del numero di round *r* dell' algoritmo: $\lim_{r\to\infty} P = 1$.

Caratteristica dell'algoritmo di consenso binario in RAPTOR è l'utilizo di una funzione random che viene invocata in alcune situazioni e che garantisce di raggiungere il consenso.

5.2.2 Presentazione e analisi dell' algoritmo utilizzato per l' implementazione del consenso binario

L'algoritmo utilizzato per l'implementazione del consenso binario, che fa uso delle primitive `broadcast` e `deliver` precedentemente introdotte, e della funzione random `coinflip()`, può essere descritto in pseudo-codice come segue (**[2]**):

```
Input: Set Of Process π
Input: Initial Proposal Value xi
Output: Decision Value d
```

```
1.    d ← ⊥                    //decision value
2.    r_i ← 0                  //round number
3.    stop_i ← ⊥               // round num where execution halts

4.    while do                 //step 1
5.    broadcast(π,x_i)
6.    Vi ← deliver(π)
7.    If ∃_{v≠⊥}:#_v(V_i)>2f+1 then
8.          x_i ← v
9.    else
10.         x_i ← ⊥
11.   end
12.   broadcast(π,x_i )              //step 2
13.   V_i ← deliver( π)
14.   If ∃_{v≠⊥}:#_v(V_i)>2f+1 then
15.         If d=⊥ then
16.               d ← v
17.               stop_i ← r_i+1
18.         end
19.         x_i ← v
20.   else if ∃_{v≠⊥}:#_v(V_i)>f+1 then
21.         x_i ← v
22.   else
23.         x_i ← coinflip()
24.   end
25.   if r_i=stop_i then
26.         halt()               //stop execution
27.   end
28.   r_i ← r_i+1
```

L‘ algoritmo è distribuito e viene eseguito da tutti i processi: ciascun processo riceve come dati di ingresso il set di processi π e il valore di proposta iniziale x_i e produce in uscita il valore di decisione *d.*

Ogni processo p_i appartenente a π comincia il protocollo inizializando *il round number* r_i a 0, il valore di decisione *d* ad un valore di default che indica che una decisione finale ancora non è stata presa, e il valore $stop_i$ ad un valore indefinito $\perp$. La variabile $stop_i$ indica il numero di round al quale l' algoritmo terminerà.

Terminata questa fase di inizializzazione il protocollo procede di round in round, e ogni round è composto da due step. Lo step 1 del protocollo (linee 5-11) è semplice: il generico processo p_i spedisce prima in broadcast a tutti il proprio valore di proposta x_i e sucessivamente salva in V_i tutti i messaggi che riceve dagli altri processi. Se in V_i ci sono almeno 2f+1 messaggi con lo stesso valore *v*, allora x_i viene settata a *v*, altrimenti viene settata a $\perp$, ad indicare che p_i non ha un proprio valore di preferenza (linee 7-11).

A questo punto comincia l'esecuzione dello step 2 (linee 12-28): ogni processo p_i esegue nuovamente broadcast di x_i e salva nuovamente in V_i tutti i messaggi che riceve dagli altri processi.

Se in V_i ci sono almeno 2f+1 messaggi con lo stesso valore *v*, allora p_i setta il valore di decisione *d* a *v*, e la variabile $stop_i$, ad indicare che l'esecuzione dell'algoritmo dovrà fermarsi al round successivo (linee 14-18). Infine x_i è aggiornato con il valore *v* (linea 19).

Se non ci sono ***2f+1*** messaggi che contengono il valore v, ma almeno f+1 allora semplicemente l' algoritmo aggiorna x_i con il valore *v*.

Infine, se nessuna delle due condizioni precedenti è rispettata, allora semplicemente si aggiorna x_i con un valore random di 0 o 1. La funzione random deve essere tale che entrambi i valori abbiano una probabilità dello 0.5 (linee 22-23).

A questo punto nell' algoritmo semplicemente si controlla se deve terminare confrontando il numero di round r_i con il valore contenuto in $stop_i$: se sono uguali l'algoritmo termina, altrimenti si incrementa il round r_i e si continua l'esecuzione del protocollo con il round successivo.

E' importante notare che, anche se non specificato direttamente in pseudo-codice, per una corretta implementazione dello stesso, bisogna garantire che il sistema sia sincrono: ciascun processo prima di eseguire la `deliver` deve essere certo che tutti gli altri processi del sistema abbiano eseguito la `broadcast` per fare in modo che la struttura dati che implementa V_i sia consistente.

Quando passeremo all' implementazione in C++ in ambiente OMNeT++, dovremo tenerne conto inserendo e implementando nel nostro sistema una qualche forma di controllo esterno che si occupi della sincronizzazione dei vari aircraft tramite lo scambio di oppurtuni messaggi di sincronizzazione.

CAPITOLO 6

IMPLEMENTAZIONE DI RAPTOR IN LINGUAGGIO C++ E AMBIENTE OMNETT++.

6.1 Modello degli aircraft

Abbiamo visto come l' airborne self separation sia un modo di concepire l' ATM nel quale è lasciata ai piloti, in real time, la possibilità di selezionare le proprie rotte, senza l' intervento di torri di controllo da terra o di altri operatori esterni. Tutto questo è reso possibile a livello fisico dall' utilizzo del sistema satellitare e della tecnologia ADS-B in sostituzione dei sistemi RADAR, a patto appunto che i singoli aircraft vengano dotati di strumentazione ADS-B a bordo. In sostanza, è necessaria l' installazione di un receiver, per ricevere il segnale GNSS che ne identifichi e ne fornisca le coordinate in tempo reale che verranno poi inviate in broadcast non solo a possibili centri di controllo ground-based, ma a tutti gli altri aircraft che si trovino ad un determinato range.

Andremo pertanto adesso ad analizzare la struttura del modello degli aircraft utilizzato in **[3]**, componente per componente, le quali verranno tradotte 1:1 in ambiente OMNeT++ tramite

linguaggio NED per la costruzione del modello, e linguaggio C per l' implementazione runtime del comportamento di queste componenti.

Nell'ordine: il modulo Wireless, per l' implementazione del physical layer e il Mac Access, il Navigator per modellare il global positioning system, l' ADS-B, e infine l' SGT, il modulo che si occuperà della gestione di possibili conflitti nella scelta delle rotte da seguire.

Prima di procedere con l' implementazione, riportiamo una breve descrizione dei componenti.

6.1.1 Componente Wireless

Il modulo wireless implementato sugli aircraft va ad implementare la radio wireless installata sui mezzi di volo, occupandosi dell' invio/ricezione del segnale a livello fisico.

All' interno del modulo sono quindi compresi due strati: il livello strettamente fisico, che gestisce la ricezione e l' invio dei pacchetti in broadcast, e il Medium Access Control, che implementa i comuni servizi di affidabilità di comunicazione.

Il modulo fornisce inoltre ai livelli superiori una funzione di broadcast che può essere usata e invocata per la gestione della comunicazione tra i vari aircraft.

Il protocollo scelto per la comunicazione wireless è lo standard IEEE 802.11, questo per maggiore semplicità visto che:

- Non esiste ancora uno standard da adottare ufficialmente per la comunicazione tra i mezzi di volo.
- L' 802.11 è già implementato in una delle librerie messe a disposizione per l'ambiente di simulazione utilizzato in questo lavoro, l' OMNeT++ (più nel dettaglio, l' implementazione

dello stesso si trova all' interno del frame work INET per OMNeT++, scaricabile liberamente all'URL *http://inet.omnetpp.org/*).

6.1.2 Il Navigator

Il modulo navigator simula il GNSS (Global Navigation Satellite System) installato sull' aircraft, nel nostro caso, il Global Positioning System (GPS) e allo stesso tempo implementa le funzioni del sistema di navigazione.

Come abbiamo visto in precedenza il GNSS si occupa di fornire all' aircraft stesso informazioni dettagliate e affidabili sulla sua posizione.

Anche l' implementazione di questo modulo sfrutta funzionalità già presenti nel simulatore OMNeT e nelle sue librerie, più precisamente utilizza i sorgenti dell'INET framework che si occupano dell' implementazione della mobilità lineare degli oggetti all' interno dell' ambiente.

Gli aircraft del nostro modello si muoveranno infatti semplicemente seguendo traiettorie in linea retta: *lo spazio simulabile è infatti bidimensionale*. Questo purtroppo è una restrizione imposta dal framework INET, che non supporta ancora spazi di simulazione tridimensionali.

A questi sorgenti sono stati aggiunti, nel lavoro cui fa riferimento **[3]**, e da cui è stato sviluppato questo lavoro di tesi, due funzionalità aggiuntive:

- Un' interfaccia per ricavare posizione, velocità, accelerazione e direzione dell' aircraft
- Un' interfaccia per aggiustare di volta in volta posizione, accelerazione e direzione per simulare le manovre del pilota.

6.1.3 L' ADS-B

Questo modulo implementa la strumentazione di bordo ADS-B necessaria all' aircraft, come visto nel capitolo precedente.

E' all'interno di esso che avviene la combinazione delle informazioni ricevuti dal GPS e quindi dal navigator, e il loro successivo invio in broadcast, tramite le primitive messe a disposizione dal servizio wireless, a tutti gli aircraft che si trovano in un determinato range prefissato. Chiameremo queste informazioni, **ADS-B messages**.

Il servizio è automatico, ossia come spiegato precedentemente, sempre attivo e senza bisogno di intervento esterno da parte del pilota.

Il modulo si trova quindi a un livello intermedio tra i livello fisico, e il modulo SGT, a cui invia le informazioni ricavate dai messaggi ADSB ricevuti dal modulo wireless, per la risoluzione e il detect di eventuali conflitti.

6.1.4 L' SGT

Al più alto livello del modello troviamo il modulo SGT che implementa tutta la logica che si occupa della detection e della risoluzione di eventuali conflitti presenti nelle informazioni scambiate dagli aircraft.

Le due funzioni principali di questo modulo sono la `rejectability` e la `selectability`. Lo spazio aereo viene riassunto a partire dalle informazioni ricevute dagli altri aircraft: la `selectability` stima i benefici che l' aircraft può avere nello scegliere determinate rotte, la `rejectability`, ne valuta i rischi di collisione.

Il modulo sfrutta la libreria *CGAL*, ottenibile all' URL *http://www.cgal.org* e scritta in C, da installare e compilare nel nostro ambiente di simulazione, che useremo per i complessi calcoli di tipo geometrico, e la libreria *dLIB* anch' essa scritta in C e ottenibile all' URL *http://dclib.sourceforge.net* utilizzata invece per la scelta della rotta ottimale.

6.2 Il Modulo RAPTOR e sua implementazione

Il modulo RAPTOR di ciascun velivolo viene costruito attraverso l'implementazione di due sottomoduli distinti: il primo RAPLogic , contiene l' implementazione della logica dell' algoritmo di consenso binario, il secondo, RAPComm, si occupa invece dell' implementazione della comunicazione del modulo RAPTOR mettendo a disposizione ad esempio le primitive di broadcast e deliver.

IMPLEMENTAZIONE DEL MODULO RAPLogic

Il modulo RAPLogic è definito come segue:

```
class RAPLogic : public cSimpleModule {
      private:
            RAPComm* comm_ptr; //pointer to the comm module, which provides
                               //the broadcast and the deliver primitives
            DataTable dt;      //collects ADSB data;
                               //the agreement will be run on these data

            int phase;
            double xi;        //initial proposal value
            int d;            // decision value
            int stop;         // round number where execution halts
            int round;
```

```
    protected:
        virtual void initialize();
        virtual void finish();
        virtual void handleMessage(cMessage* msg_ptr);
        virtual int agreement_start (int xi, int num);
        virtual int agreement(int num);
        virtual void dispatchPacket(cMessage* msg_ptr);
        virtual void storeData(ADSBDataMessage* dataMsg_ptr);
        virtual RAPTable deliver();
        virtual void broadcast(int round, int xi);
        virtual void sync(int num);
        virtual void advanceRound();
};
```

Sono presenti le dichiarazioni di alcune variabili utili per l'implementazione dell'algoritmo di consenso binario che corrispondono uno a uno, a quelle che abbiamo presentato precedentemente durante la stesura in pseudocodice dell' algoritmo: x_i, `stop`, `d`, e `round`.

Assieme a queste, sono dichiarate due variabili di supporto: il puntatore `comm_ptr` per l' accesso al modulo RAPComm (ci serviranno le primitive `broadcast` e `deliver`) e la struttura dati `dt` che implementa il vettore dove vengono memorizzate le informazione ricevute da ciascun aircraft dopo ogni broadcast.

Analizziamo il comportamento del modulo attraverso la descrzioni dei suoi metodi.

La funzione `initialize()` è il costruttore del modulo, viene invocata alla sua creazione e si occupa dell' inizializzazione dello stesso.

```
void RAPLogic::initialize() {
	phase = 0;
	round= 0;
	xi=1;
	d = NaDecision;
	stop = NaDecision;
	comm_ptr = dynamic_cast<RAPComm*>(parentModule()->submodule("comm"));
	if (comm_ptr == NULL){ error("Could not find the comm module"); }
	dt.allocDataTable(parentModule()->parentModule()
	                                  ->parentModule()->par("numHosts"));
}
```

Questa funzione semplicemente inizializza le variabili così come avevamo visto nella definizione dell' algoritmo e alloca la struttura dati `dt`, attraverso il metodo `allocDataTable()`.Una volta creato, il modulo resta in attesa della ricezione di un messaggio che verrà catturato dal metodo `handleMessage()`. Tutte le volte che un messaggio viene inviato al modulo RAPLogic, questo viene raccolto e gestito dalla funzione `handleMessage`:

```
void RAPLogic::handleMessage(cMessage* msg_ptr) {

	ev<<"RAPLogic: received "<<msg_ptr->name()<<" message"<<endl;

	int n = parentModule()->parentModule()->parentModule()
	                                  ->par("numHosts");//number of process
	if (strcmp(msg_ptr->name(), "signal") == 0)  {
		ev<<" signal arrived"<<endl;
		int res=agreement(n);
		if (res==-2) agreement_start(xi,n);
	}
	else if(strcmp(msg_ptr->name(), "agreement_start")==0)
```

```
                                                    agreement_start(xi,n);
        else if(dynamic_cast<ADSBDataMessage*>(msg_ptr) != NULL) {
//collects ADSB data. The agreement protocol will be executed on these data
                storeData((ADSBDataMessage*)msg_ptr);
                double position =(dynamic_cast<ADSBDataMessage*>(msg_ptr))
                                                    ->getPosition().x;
                int ID= (dynamic_cast<ADSBDataMessage*>(msg_ptr))
                                                    ->getAircraftID();
                ev<<" position of aircraft num "<<ID<<"is "<<position<<endl;
                //binary consensus, example Threshold = 150
                xi= (((dynamic_cast<ADSBDataMessage*>(msg_ptr))
                                                ->getPosition()).x)>100?1:0;
                agreement_start(xi,n);
                //dispatching the received packet to SGT
                dispatchPacket(msg_ptr);
        }
        else{
                ev<<"RAPLogic: received unknown message type"<<endl;
                delete msg_ptr;
        }
}
```

I messaggi che un modulo RAPLogic può ricevere sono di due tipi: o un messaggio ADSB da parte di altri aircraft, o un messaggio di sincronizzazione durante l' esecuzione del protocollo, da parte del RAPCoordinator.

Se il messaggio è una signal da parte del RAPCoordinator, semplicemente continua l' esecuzione del protocollo secondo le modalità definiti più avanti.

Alla ricezione invece di un messaggio di tipo ADSB, si collezionano le informazioni ricevute in esso (attraverso la `storeData`), sulla base di queste si setta il valore di proposta `xi` e si inizia l' esecuzione del protocollo di agreement invocando la `agreement_start(xi,n)`.

```
int RAPLogic::agreement_start (int xi, int num) {
	ev<<"starting agreement with xi= "<<xi<<" id= "<<id()<<endl;
	broadcast(2*phase+1,xi);
	ev<<"calling sync"<<endl;
	sync(num);
	return 0;
}
```

L' esecuzione del protocollo inizia con l' invio del valore di proposta x_i in broadcast, poi il processo invia un messaggio di sincronizzazione al `RAPCoordinator` e si mette in attesa di una signal da parte di quest' ultimo, che verrà spedita quanto tutti i processi avranno eseguito la loro broadcast.

Nel seguito riportiamo la logica del modulino `RAPCoordinator`, che implementa la sincronizzazione richiesta in modo molto semplice:

```
void RAPCoordinator::handleMessage(cMessage* msg_ptr) {
	int num=parentModule()->par("numHosts");
	const char* name=msg_ptr->name();
	ev<<name<<endl;

	if (strcmp(name,"sync")==0) {
		count++;
		ev<<"updating counter in RapCoordinator "<<count<<endl;
		if (count == num){
			cMessage* signal= new cMessage("signal");
```

```
                for (int i=0;i<num;i++){
                //broadcasting signal message to all hosts
                cModule *destinationModule= parentModule()
                                                ->submodule("host",i);
                cMessage* copy= (cMessage*) signal->dup();
                double delay=truncnormal (0.0,0.2);
                sendDirect(copy,delay,destinationModule,"rapSyncIn");
                }
            delete signal;
            count=0;
            }
      }
}
```

RAPCoordinator incrementa la variabile count alla ricezione di una sync(), e se count ha raggiunto un valore pari al numero di processi dell' applicazione, allora invia un messaggio di signal a tutti i processi e riazzera il contatore.

Una volta ricevute le signal, come visto prima, i processi iniziano l' esecuzione del protocollo vero e proprio, implementato dalla funzione agreement().

Questa funzione è strutturata attraverso uno switch i cui due case , rappresentano i due step dell' algoritmo che compongono ogni round.

Poiché è stata appena ricevuta una signal, siamo sicuri che tutti i processi hanno eseguito la broadcast e possiamo quindi invocare la funzione deliver() per memorizzare in rt tutti i messaggi ricevuti con i valori di proposta di decisione degli altri processi.

Nel vettore num_of_occurence andiamo a memorizzare le occorenze di ciascun valore ricevuto: se una occorenza di queste supera la soglia (2*F+1), x_i viene aggiornato di conseguenza. Lo step

1 termina aggiornando il round e inviando in broadcast x_i (seguita come sempre da un messaggio di sync())

```
int RAPLogic::agreement(int n) {
	int deliver_round= round%2;
	RAPTable rt;
	int* dataTab;
	int num_of_occurence[CARDINALITY];
	//initializing
	for (int i=0; i<CARDINALITY;i++) num_of_occurence[i]=0;
	ev<<"agreement phase "<<phase<<endl;
	ev<<"agreement round "<<round<<endl;
	switch(deliver_round) {
		case 0:	ev<<"executing switch case 0"<<endl;
			rt=deliver();
			dataTab= new int [rt.size()];
			rt.getdataTab(dataTab);
	//counting number of occurence of every value in dataTab and storing
	  them
			for (int i=0; i<CARDINALITY;i++){
				for (int j=0; j< n; j++){
					if(dataTab[j]==i) num_of_occurence[i]++;
					}
			}
			xi= NaDecision;
			for (int i=0; i<CARDINALITY; i++)
				if (num_of_occurence[i]>= (2*F+1)) { xi=i;
									break;}
			delete dataTab;
			advanceRound();
```

```
                broadcast(2*phase+2,xi);
                sync(n);
                break;
        case 1:
                ...
                ...
                ...
}
```

Una volta che tutti i processi avranno eseguito lo step 1 e la `broadcast`, la successiva signal del `RAPCoordinator` porterà il modulo `RAPLogic` a richiamare nuovamente la funzione `agreement()`, di cui verrà eseguito stavolta lo step 2.

```
int RAPLogic::agreement(int n) {
      ...
      ...
      switch(deliver_round) {
            case 0:     ...
                        ...
                        ...
                        break;

            case 1:
                        ev<<"executing switch case 1"<<endl;
                        rt=deliver();
                        dataTab= new int [rt.size()];
                        rt.getdataTab(dataTab);           //getting dataTab
```

```
//counting number of occurence of every value in dataTab
    for (int i=0; i<CARDINALITY;i++){
        for (int j=0; j< n; j++){
            if(dataTab[j]==i) num_of_occurence[i]++;
            }
    }
    for (int i=0; i<CARDINALITY; i++) {
        if (num_of_occurence[i]>=(2*F+1)) {
            xi=i;
            if (d==NaDecision) {
                d=i;
                stop=phase+1;
                break;
                }
        }
    }
    if (d == NaDecision) {
        for (int i=0; i<CARDINALITY; i++) {
            if (num_of_occurence[i]>=(F+1)){
                xi=i;
                break;
                }
            else xi= rand()% CARDINALITY;
            ev<<"xi=coin_flip()= "<<xi;
        }
    }
    ev<<" phase = "<<phase<<" stop= "<<stop<<endl;
    advanceRound();
```

```
                        if (stop==phase) {
                                    ev<<" AGREEMENT REACHED IS"<<d<<endl;
                                    return d ;
                                    }
                        phase++;
                        ev<< "repeat cicle"<<endl;
                        return -2;              // -2= repeat cycle
        }
        return -1;
}
```

Ricaviamo la tabella con le informazioni ricevute (`deliver()`), contiamo le occorrenze di ciascun valore memorizzandole nel vettore `num_of_occurence`, e passiamo a eseguire un primo controllo, per vedere se cè un valore la cui occorrenza supera la soglia `2f+1`: nel qual caso abbiamo raggiunto la nostra decisione, viene settata la variabile `stop`, usciamo dallo switch e la funzione al round successivo ritornerà il valore di decisione *d*.

Se il primo *if* non è soddisfatto, si esegue un altro controllo per vedere se nella tabella ottenuta c'è almeno un valore *i* (0 o 1 nel caso di consenso binario), la cui occorenza superi la soglia di `f+1`: se questo è verificato, il nuovo valore proposto dal processo sarà proprio *i*, altrimenti il nuovo valore da proporre in broadcast è deciso tramite la funzione `coinflip` (funzione `rand()`).

A questo punto semplicemente viene incrementato il valore della variabile `phase` e la funzione ritorna -2 ad indicare che un agreement non è stato ancora raggiunto e che l' esecuzione del protocollo deve proseguire dalla funzione `agreeement_start()`.

IMPLEMENTAZIONE DEL MODULO RAPComm

Il sottomodulo RAPComm è molto semplice, e si occupa della trasmissione broadcast a livello MAC delle informazioni necessarie alla logica del protocollo.

```
class RAPComm : public cSimpleModule {
	private:
		RAPTable rt;  // collects Agreement messages
		protected:
		virtual void initialize();
		virtual void finish();
		virtual void handleMessage(cMessage* msg_ptr);
	public:

		virtual void broadcast(int val); // broadcasts a value
		virtual void deliver(RAPTable& rt);
};
```

Il suo unico campo privato è rappresentato dalla struttura dati `RAPTable rt`, (la presenteremo successivamente nel dettaglio) che implementa la tabella RAPTOR dove vengono collezionati i valori scambiati ad ogni ciclo di broadcast.

La funzione `handleMessage()` del modulo è definita come segue:

```
void RAPComm::handleMessage(cMessage* msg_ptr) {
	ev<<"RAPComm: received "<<msg_ptr->name()<<" message"<<endl;
	if(dynamic_cast<RAPDataMessage*>(msg_ptr) != NULL) {
		// collects the received messages
		RAPDataMessage* dataMsg_ptr = (RAPDataMessage*)msg_ptr;
		int id = dataMsg_ptr->getAircraftID();
		int val = dataMsg_ptr->getVal();
```

```
            rt.setVal(id, val);
            rt.printTable();
            delete msg_ptr;
      }

      else{
            ev<<"dispatching message"<<msg_ptr->name()<<"to RAP logic"<<end;
            send(msg_ptr, "dispatch");
      }
}
```

Semplicemente, tutte le volte che il modulo riceve un messaggio, ne estrae le informazioni contenute (identificatore dell' aircraft sorgente e valore spedito) , le inserisce nella struttura dati `rt` e ne stampa in output il contenuto.

La funzione `deliver()` è la seguente:

```
void RAPComm::deliver(RAPTable& rt) {
      Enter_Method("RAPComm::deliver()");
      rt = this->rt;
      rt.flushRAPTable();
}
```

La prima riga di codice è la sintassi OMNeT++ per rendere visibili a moduli esterni, il metodo che stiamo definendo. Per il resto la `deliver` non fa altro che copiare nell' argormento passato per riferimento, la tabella `rt` attuale: fatto ciò, la cancella (una volta acquisite, le informazioni sono obsolete, non cè motivo di tenerne traccia).

Infine, ecco l' implementazione della `broadcast:`

```
void RAPComm::broadcast(int val) {
	Enter_Method_("RAPComm::broadcast()");
	static MACAddress broadcastAddress("ff:ff:ff:ff:ff:ff");
	// creates the message
	RAPDataMessage* dmsg_ptr = new RAPDataMessage("RAPDataMessage");
	dmsg_ptr->setVal(val);
	dmsg_ptr->setAircraftID (id());
	rt.setVal(id(), val);
	// adds control info
	Ieee802Ctrl *controlInfo = new Ieee802Ctrl();
	controlInfo->setDest(broadcastAddress);
	dmsg_ptr->setControlInfo(controlInfo);
	// sends out
	double delay= truncnormal (0.0,0.2);
	ev<<"sending rap message with val="<<val<<endl;
	int ret=sendDelayed(dmsg_ptr,delay, "RAPOut");
	if (ret==0) ev<<"success broadcast"<<endl;
}
```

Ogni aircraft setta nella RAPTable le informazioni che sta per spedire in broadcast a tutti gli altri, poi comincia la costruzione del messaggio che oltre a un corpo (id e valore proposto) necessita anche di alcune informazioni di controllo, tra le quali ad esempio l' indirizzo MAC di destinazione (nel nostro caso caso, il *broadcast address* con tutti i 48 bit settati a 1).

E' importante notare che i broadcast di ciascun processo sono appositamente e volutamente ritardate tramite un ritardo random uniformente distribuito all' interno della gaussiana troncata tra 0 e 0.2sec:

```
double delay= truncnormal (0.0,0.2);
```

Questo si è reso necessario ai fini implementativi, per impedire che si verificasse l'invio simultaneo di più RAPMessage a livello MAC, con possibilità di perdita dei messaggi stessi da parte del receiver.

6.3 La struttra dati RAPTable

Come già specificato, la struttura dati RAPTable implementa la tabella dove vengono collezzionati i valori proposti dai processi di volta in volta: è costituita da una coppia di vettori statici di interi: `dataTab[]` memorizza i valori proposti, `idTab[]` gli ID degli aircraft coinvolti.

L' associazione *Id-Valore* è ricavata in questo modo: il valore proposto dall' aircraft identificato dall' ID contenuto `idTab[i]`, è contenuto esattamente in `dataTab[i]`.

```
class RAPTable{
      protected:
            int* dataTab; // collects data of other aircraft
            int* idTab;   // associates the aircraft ID to indexes
            int maxDim;
            int currDim;

      public:
            // allocates an empty table
            RAPTable() { dataTab = 0; idTab = 0; maxDim = currDim = 0; }
            //allocates a table with n entries
            virtual void allocRAPTable(int n);
            //invalidates table entries
            virtual void flushRAPTable() { currDim = 0; }
            RAPTable(const RAPTable& bt);
            RAPTable& operator=(const RAPTable& dt);
```

```
        int size() const { return currDim; }
        virtual const int* getAircraftIDs() const;
        virtual void getdataTab(int p[]);
        virtual bool setVal(int aircraftID, int val);
        virtual bool getVal(int aircraftID, int& val) const;
        void printTable() const;
        virtual ~RAPTable() { delete[] dataTab; delete[] idTab; }

};
```

Le funzioni sono molto semplici e autoesplicative, si tratta di metodi che lavorano su array di interi per poter allocare e eliminare i vettori, ottenere e settare coppie id-valori, ottenere la lista degli id degli aircraft, la lista dei valori proposti etc.

6.4 Multivalued Consensus: estensione

Estendiamo ora l' algoritmo di consenso binario per ottenere un consenso multivalore in modo che i processi possano accordarsi su un qualsiasi valore v di un dato dominio V: accelerazione, velocità, direzione, posizione di un dato aircraft.

L'algoritmo binario precedentemente implementato sarà una primitiva su cui l'algoritmo di consenso multivalore va ad appoggiarsi **[2]**.

In fase di progetto si sceglierà un opportuno valore di default su cui i processi si accorderanno nel caso in cui non riescano a raggiungere un agreement fra i vari valori proposti dai singoli processi. Nel dettaglio i processi si accorderanno su un valore v se questo è stato proposto da un numero di processi superiore a f+1 , altrimenti sul valore di default.

Anche questo algoritmo gode come il precedente delle proprietà tipiche di un algoritmo di agreement: nessuna coppia processi deciderà valori differenti, e tutti i processi giungeranno alla fine a una decisione, quale essa sia (terminazione).

L' algoritmo, presentato formalmente in **[2]**, alla cui dimostrazione rimandiamo, è il seguente:

Input: Initial Proposal Value x_i

Output: Decision Value d

1. broadcast(1,π,x_i) //step 1
2. $V_i \leftarrow$ deliver {1,π)
3. **If** $\exists_{v\neq\perp}:\#_v(V_i)>2f+1$ **then**
4. $x_i \leftarrow v$
5. **else**
6. $x_i \leftarrow \perp$
7. **end**
8. broadcast(2, π,x_i) //step 2
9. $V_i \leftarrow$ deliver (1,π)
10. **If** $\exists_{v\neq\perp}:\#_v(V_i)>2f+1$ **then**
11. $b_i \leftarrow 1$
12. **else**
13. $b_i \leftarrow 0$
14. **end**
15. $c_i \leftarrow$ binary_consensus(b_i) //step 3
16. **if** $c_i=1$ **then**
17. $d_i \leftarrow v:\#_{v\neq\perp}(V_i)>f+1$
18. **else**
19. $d_i \leftarrow \perp$
20. **end**

L' algoritmo si suddivide in 3 parti, separate tra loro da coppie di broadcast-receive.

Nella prima parte, ogni processo invia a tutti gli altri la propria proposta e memorizza a sua volta i valori che riceve: se tra questi valori ve ne è uno *v* che supera per occorrenze la soglia 2f+1, ogni processo aggiorna la propria proposta con quel valore *v*.

Se non cè nessun valore tra quelli ricevuti che soddisfa questa condizione, il processo aggiorna *xi* con il valore di default.

A questo punto avviene il secondo scambio di messaggi (broadcast) e successiva receive di ogni processo dei valori di proposta trasmessi dagli altri.

Questo secondo passo, prepara il valore di proposta per l' esecuzione dell' algoritmo di consenso binario che verrà eseguito nel terzo e ultimo passo: se un processo riceve almeno 2f+1 valori diversi da quello di default, allora setta il proprio valore di proposta binario pari 1, altrimenti a 0.

Nel terzo e ultimo passo viene richiamato l' algoritmo di consenso binario sui valori b_i di ciascun processo: se i processi si accordano sul valore finale di 1 allora i processi decideranno sul valore v che appare almeno f+1 volte in V_i, altrimenti se l' algoritmo di consenso binario restituisce 0, essi si accordano sul valore di deafult prestabilito.

CAPITOLO 7

SIMULAZIONI DI RAPTOR

Come detto precedentemente il protocollo scelto su cui si basano le simulazioni e la comunicazione tra i vari aircraft è il protocollo wireless standard 802.11, le cui funzioni troviamo già implementate e disponibili all' interno del framework INET-svn.

Tutti i file necessari alle varie run di simulazione sono contenuti nella directory "Omnet++/mysamples/sgt-inet", in particolare:

1. il file ned "net80211.ned", che contiene la descrizione del modello comune a tutte le run.
2. l' eseguibile "runme.bat", prodotto dalla compilazione di OMNeT++ con i nostri sorgenti
3. il file di configurazione "Omnetpp.ini", che contiene tutti i parametri di inizializzazione tramite i quali configurare le run.

7.1 Il modello di rete: il file `net80211.ned`

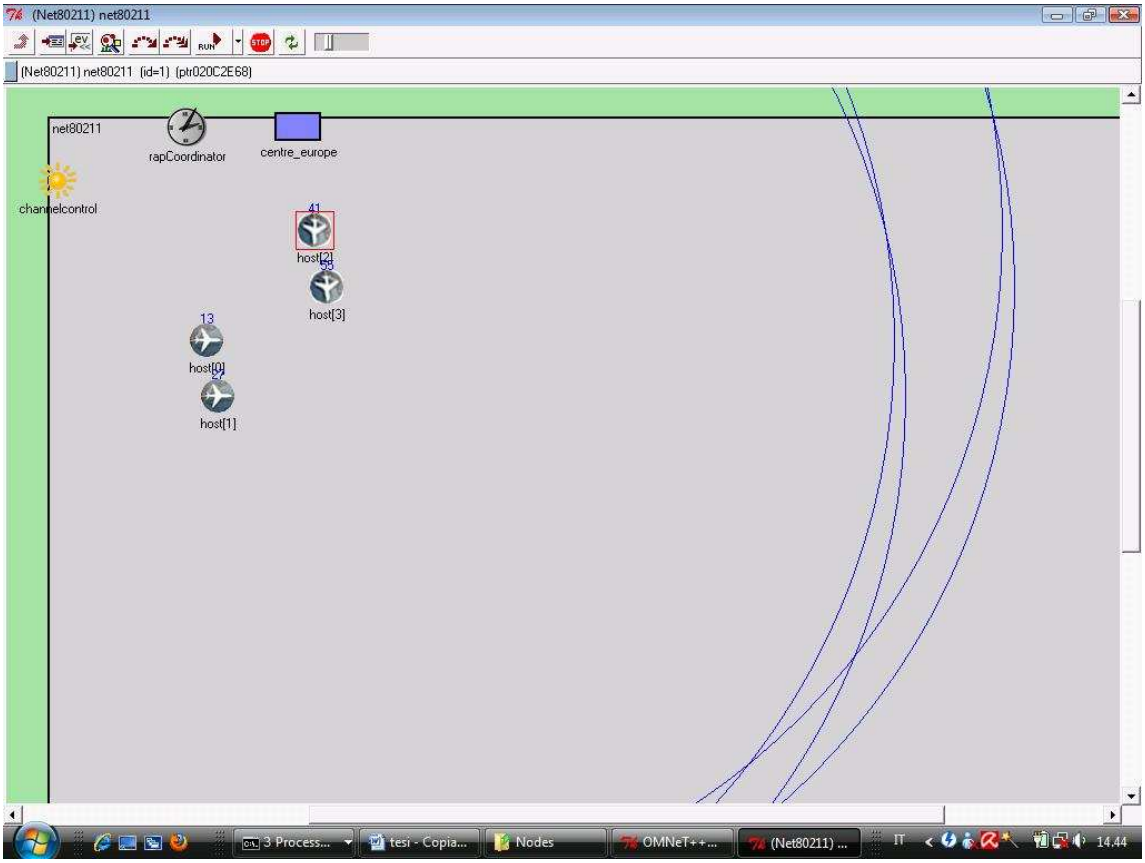
Vediamo brevemente il file "net80211.ned" che mostra la struttura della rete:

```
import
    "SGTMobileHost",
    "ChannelControl",
    "RAPCoordinator";

module Net80211
    parameters:
        numHosts: numeric const,
        numFaulty: numeric const,
        playgroundSizeX: numeric const,
        playgroundSizeY: numeric const;

    submodules:

        centre_europe: Map;
            display: "p=224,8;b=40,24";

        host: SGTMobileHost[numHosts];
            display: "i=aircraft/aircraft_30x30;r=,,blue";

        channelcontrol: ChannelControl;
            parameters:
                playgroundSizeX = playgroundSizeX,
                playgroundSizeY = playgroundSizeY;
            display: "p=8,56;i=misc/sun";

        rapCoordinator: RAPCoordinator;
            gatesizes: rapSyncIn[numHosts],rapSyncOut[numHosts];
            display: "p=124,8;i=device/clock";

    connections nocheck:
            display: "b=252,100";
endmodule
```

Come si può notare l' architettura di base è, almeno concettualmente, semplice: la nostra rete è composta da un canale di comunicazione `ChannelControl`, un `RAPCoordinator`, il modulo che si occuperà di gestire le sincronizzazioni tra i vari aerei coinvolti e da un certo numero di hosts (variabile e configurabile) `SGTMobileHost[numHosts]`. Ogni modulo `SGTMobileHost` corrisponde ad un singolo aereo e ne descrive la struttura.

Questi singoli componenti, importati tramite direttiva `import` sono poi descritti nel dettaglio dai file NED all' interno della cartella `unipi` che contiene tutti i sorgenti del lavoro.

Un esempio dello scenario che si ottiene utilizzando 4 aerei è il seguente: le circonferenze rappresentano la copertura del segnale wireless di ciascun aircraft, aumentando o diminuendo la potenza del segnale trasmesso è possibile coprire aree più o meno estese.



7.1: esempio di scenario con 4 aircraft

7.2 Configurare le simulazioni: il file `Omnetpp.ini`

All' interno della directory di simulazione si trova il file Omnetpp.ini: è il file di configurazione, nei quali troviamo la descrizione di tutte i run che vogliamo eseguire. Possiamo modificare i parametri di tutto ciò che caratterizza una simulazione: il numero degli host, il numero di quelli faulty, la loro

posizione iniziale, la velocità, la destinazione, i parametri del canale wireless e quelli del ricevitore radio installato su ogni aircraft.

Per quanto riguarda questi ultimi, sono uguali per ogni simulazione, sia per semplicità, sia perché la validità del protocollo RAPTOR implementato non dipende da tali parametri.

```
wireless.nic.radio.bitrate=1E+6 ; // 1Mbps
wireless.nic.radio.snirThreshold = 4  // in dB
```

Abbiamo un canale wireless in grado di trasmettere a 1Mbps e in grado di ricevere segnali che presentino un rapporto segnale/rumore di almeno 4db. Se un qualsiasi messaggio dovesse arrivare a destinazione con una soglia inferiore, verrebbe scartato e trattato come rumore.

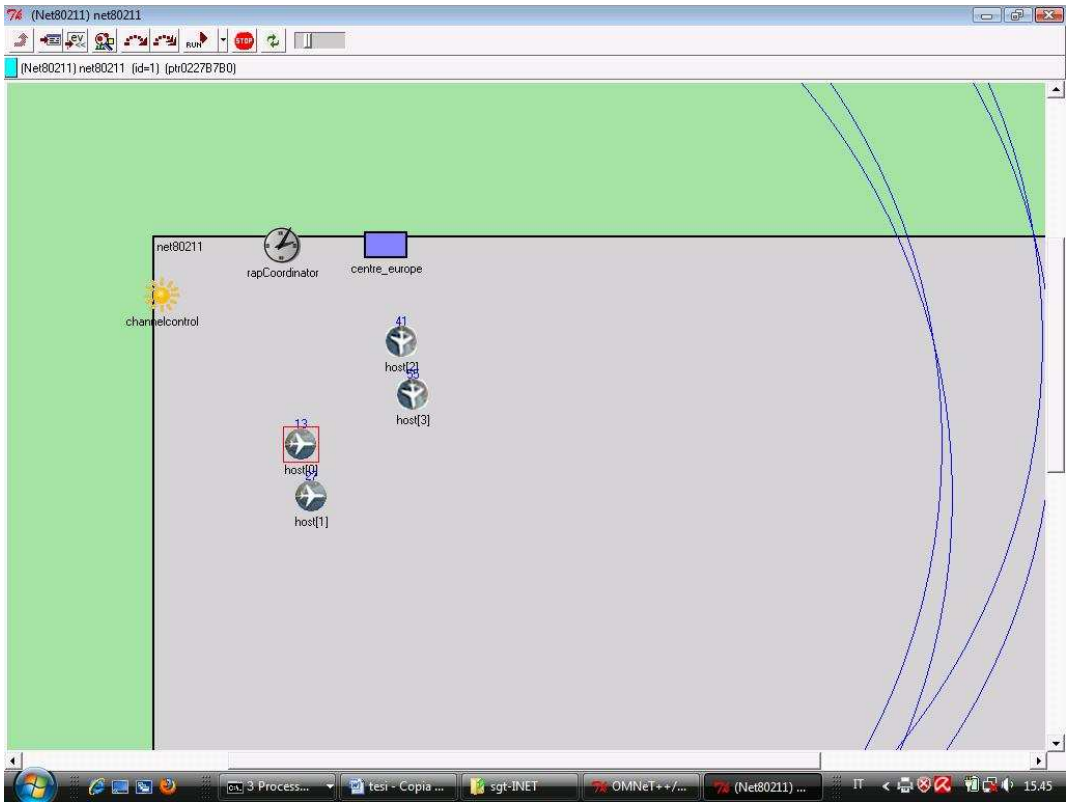
Per il rumore termico, la sensitività, la frequenza portante, e gli altri parametri strettamente riguardanti il "*physical layer*" sono stati scelti i valori di default consigliati nella documentazione del framework INET, visto che comunque non hanno particolare importanza a fini della valutazione del protocollo.

```
wireless.nic.radio.transmitterPower=2.0  // in milliwatt
wireless.nic.radio.carrierFrequency=1.090e+9 //1090MHz -- 2.4E+9
wireless.nic.radio.thermalNoise=-110
wireless.nic.radio.sensitivity=-85
```

Una volta scelti i parametri del ricevitore radio (potenza del segnale, soglia minima di SNR, sensitività, bit-rate, rumore termico etc) e del canale wireless, scrivere un run è abbastanza semplice, basta settare il numero totale di host, il numero di quelli faulty e la loro posizione di partenza/destinazione sullo scenario.

7.3 Esecuzione del protocollo RAPTOR implementato

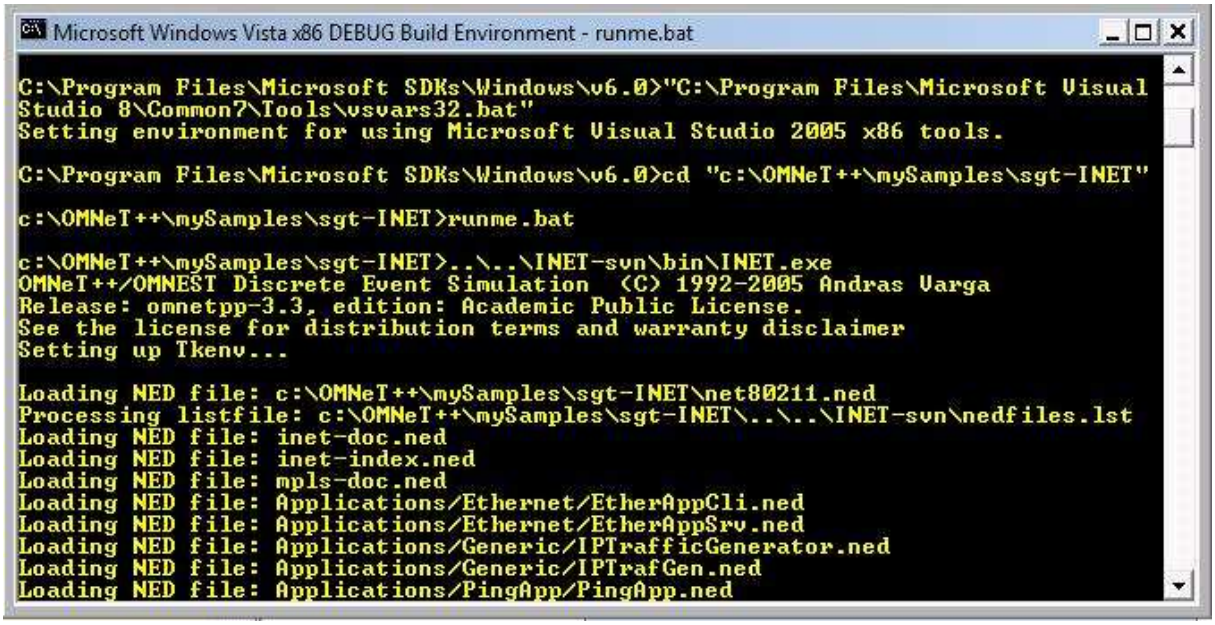
7.3.1 Scenario 1: 4 host, 1 processo faulty.



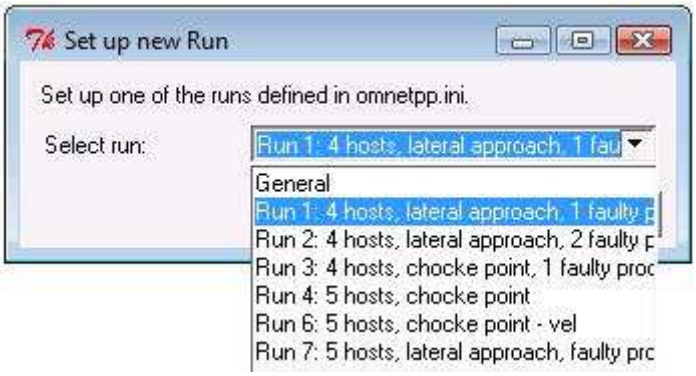
Esaminiamo dapprima un caso semplice con 4 host, di cui uno faulty. La condizione N>=3m+1 è soddisfatta.

- Il protocollo di agreement avverrà sulla coordinata **X** trasmessa dagli aircraft nel messaggio ADSB assieme a tutte le altre. Trattandosi di algoritmo binario è stata usata la convenzione che se la coordinata x comunicata è maggiore di 100 pixel, allora x_i=1, altrimenti x_i=0.
- Per quanto riguarda il processo faulty, è stato implementato semplicemente facendo in modo che invii sempre x_i=0, indipendentemente dalle informazioni che ricevono dagli altri aircraft durante il protocollo. Non seguono correttamente il protocollo.

Spostiamoci nella directory dell' eseguibile e avviamo il simulatore:



Caricati tutti i file NED sorgenti, si attiverà un menù a tendina tramite il quale è possibile scegliere a runtime quale simulazione lanciare tra quelle che il simulatore trova descritte nel file di configurazione `Omnetpp.ini`. Scegliamo la prima tra le disponibili:



Facciamo partire la simulazione: il processo sorgente iniziale sarà quello relativo alla logica installata sull' aircraft 0, il primo aereo che inizia la navigazione e di conseguenza comunica per primo agli altri processi la sua posizione iniziale:

```
** Event #0.  T=0.0000000  ( 0.00s).  Module #10 `net80211.host[0].sgt'
Setting direction to 0
== Aircraft13 starting up navigation
** Event #1.  T=0.0000000  ( 0.00s).  Module #24 `net80211.host[1].sgt'
Setting direction to 0
== Aircraft27 starting up navigation
** Event #2.  T=0.0000000  ( 0.00s).  Module #38 `net80211.host[2].sgt'
Setting direction to 1.72945
== Aircraft41 starting up navigation
** Event #3.  T=0.0000000  ( 0.00s).  Module #52 `net80211.host[3].sgt'
Setting direction to 1.5708
== Aircraft55 starting up navigation
```

All' istante t=16s, evento #117, chiamiamolo T_i nel tempo di simulazione, abbiamo il trigger del timer ADSB: il messaggio ADSB contenente le informazioni sull' aircraft 0 viene inviato in broadcast a tutti gli aircraft nel raggio di ricezione, e ha inizio il protocollo di agreement.

```
** Event #115.  T=     16  (16.00s).  Module #11 `net80211.host[0].adsb'
handling message ADSB_TIMER
sending new adsb message
** Event #116.  T=     16  (16.00s).  Module #19 `net80211.host[0].wireless.sendSplitter'
sendersplitter: sended ADSBDataMessage message
** Event #117.  T=     16  (16.00s).  Module #15 `net80211.host[0].rap.logic'
RAPLogic: received agreement_start message
starting agreement with xi= 1, phase= 0 id= 15
setting id e val of rap data message equal to16and1
sending rap message with val=1
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

Tutti gli altri hosts che si trovano all' interno del raggio di comunicazione wireless ricevono i messaggi ADSB della sorgente, iniziando così anche loro il protocollo di agreement.

Host 1:

```
** Event #137.  T=16.001186  (16.00s).  Module #29 `net80211.host[1].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 13: 156,200
(RAPLogic) received adsb-in position data from 13 pos:(156,200)
received destination from aircraft 13: 400,200
(RAPLogic) received adsb-in destination data from 13 dst:(400,200)
received velocity from aircraft 13: 1
(RAPLogic) received adsb-in velocity data from 13 vel:1
received direction from aircraft 13: 0
(RAPLogic) received adsb-in direction data from 13 dir:1
 position of aircraft num 13is 156
starting agreement with xi= 1, phase= 0 id= 29
setting id e val of rap data message equal to30and1
sending rap message with val=1
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

Host 3:

```
** Event #145.  T=16.001186  (16.00s).  Module #58 `net80211.host[3].rap.comm'
RAPComm: received ADSBDataMessage message
dispatching message ADSBDataMessage to RAP logic
** Event #146.  T=16.001186  (16.00s).  Module #57 `net80211.host[3].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 13: 156,200
(RAPLogic) received adsb-in position data from 13 pos:(156,200)
received destination from aircraft 13: 400,200
(RAPLogic) received adsb-in destination data from 13 dst:(400,200)
received velocity from aircraft 13: 1
(RAPLogic) received adsb-in velocity data from 13 vel:1
received direction from aircraft 13: 0
(RAPLogic) received adsb-in direction data from 13 dir:1
 position of aircraft num 13is 156
starting agreement with xi= 1, phase= 0 id= 57
setting id e val of rap data message equal to58and1
sending rap message with val=1
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

Host 2:

```
RAPComm: received ADSBDataMessage message
dispatching message ADSBDataMessage to RAP logic
** Event #155.  T=16.001186  (16.00s).  Module #43 `net80211.host[2].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 13: 156,200
(RAPLogic) received adsb-in position data from 13 pos:(156,200)
received destination from aircraft 13: 400,200
(RAPLogic) received adsb-in destination data from 13 dst:(400,200)
received velocity from aircraft 13: 1
(RAPLogic) received adsb-in velocity data from 13 vel:1
received direction from aircraft 13: 0
(RAPLogic) received adsb-in direction data from 13 dir:1
 position of aircraft num 13is 156
starting agreement with xi= 1, phase= 0 id= 43
I'm Faulty
setting id e val of rap data message equal to44and0
sending rap message with val=0
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

Come possiamo notare l' Host 2 è il processo Faulty che non seguirà le indicazioni del protocollo.

Il protocollo ha termine dall' **evento #644,** quando gli host raggiungono correttamente l' agreement sul valore 1, il valore corretto che ci aspettavamo:

```
** Event #644.  T=17.211446  (17.21s).  Module #43 `net80211.host[2].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:44 val:0
aircraftID:58 val:1
aircraftID:16 val:1
aircraftID:30 val:1
---
 numoccurence= 1 i = 0
 numoccurence= 3 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #645.  T=17.23519  (17.23s).  Module #57 `net80211.host[3].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:58 val:1
aircraftID:44 val:0
aircraftID:16 val:1
aircraftID:30 val:1
---
 numoccurence= 1 i = 0
 numoccurence= 3 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #647.  T=17.416122  (17.41s).  Module #29 `net80211.host[1].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:30 val:1
aircraftID:58 val:1
aircraftID:44 val:0
aircraftID:16 val:1
---
 numoccurence= 1 i = 0
 numoccurence= 3 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

Il tempo di esecuzione totale del protocollo con 4 host è pertanto:

$$\boldsymbol{Tempo\ di\ esecuzione\ totale} = \boldsymbol{Tf} - \boldsymbol{Ti} = 17.416122 - 16 = \mathbf{1.4}\ \text{secondi.}$$

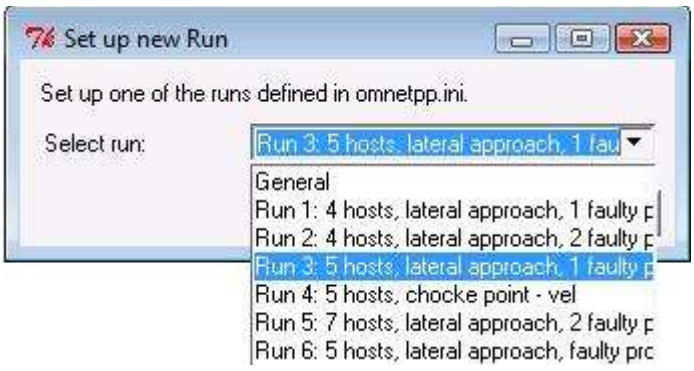
Il numero totale di pacchetti RAP, che possiamo ottenere consultando il file di output dove abbiamo scelto di registrare i messaggi RAP scambiati, ovvero il volume di traffico del protocollo è di 12 per ogni host, per un totale di 48.

$$\boldsymbol{Numero\ totale\ di\ messaggi\ RAP\ scambiati} = 12 * 4 = \mathbf{48}$$

7.3.2 Scenario 2: 5 host, 1 processo faulty.

Essendo M=1 e N=5, la condizione $\boldsymbol{N} >= \boldsymbol{3M} + \mathbf{1}$, è soddisfatta.

- Il protocollo di agreement avverrà sulla coordinata **X** trasmessa dagli aircraft nel messaggio ADSB assieme a tutte le altre. Trattandosi di algoritmo binario è stata usata la convenzione che se la coordinata x comunicata è maggiore di 100 pixel, allora x_i=1, altrimenti x_i=0.
- Per quanto riguarda il processo faulty, è stato implementato semplicemente facendo in modo che invii sempre un valore random di x_i, indipendentemente dalle informazioni che ricevono dagli altri aircraft durante il protocollo. Non segue correttamente il protocollo.



Processo sorgente: host #0, aircraft_id=14:

```
** Event #0.  T=0.0000000  ( 0.00s).  Module #11 `net80211.host[0].sgt'
Setting direction to 0
== Aircraft14 starting up navigation
** Event #1.  T=0.0000000  ( 0.00s).  Module #25 `net80211.host[1].sgt'
Setting direction to 0
== Aircraft28 starting up navigation
** Event #2.  T=0.0000000  ( 0.00s).  Module #39 `net80211.host[2].sgt'
Setting direction to 1.72945
== Aircraft42 starting up navigation
** Event #3.  T=0.0000000  ( 0.00s).  Module #53 `net80211.host[3].sgt'
Setting direction to 1.5708
== Aircraft56 starting up navigation
** Event #4.  T=0.0000000  ( 0.00s).  Module #67 `net80211.host[4].sgt'
Setting direction to 3.14159
== Aircraft70 starting up navigation
```

$Ti = 16\ secondi$, evento #142

```
** Event #140.  T=     16  (16.00s).  Module #12 `net80211.host[0].adsb'
handling message ADSB_TIMER
sending new adsb message
** Event #141.  T=     16  (16.00s).  Module #20 `net80211.host[0].wireless.sendSplitter'
sendersplitter: sended ADSBDataMessage message
** Event #142.  T=     16  (16.00s).  Module #16 `net80211.host[0].rap.logic'
RAPLogic: received agreement_start message
starting agreement with xi= 1, phase= 0 id= 16
setting id e val of rap data message equal to17and1
sending rap message with val=1
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

Faulty process: Host #3

```
** Event #172.  T=16.001186  (16.00s).  Module #58 `net80211.host[3].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 14: 156,200
(RAPLogic) received adsb-in position data from 14 pos:(156,200)
received destination from aircraft 14: 400,200
(RAPLogic) received adsb-in destination data from 14 dst:(400,200)
received velocity from aircraft 14: 1
(RAPLogic) received adsb-in velocity data from 14 vel:1
received direction from aircraft 14: 0
(RAPLogic) received adsb-in direction data from 14 dir:1
 position of aircraft num 14is 156
starting agreement with xi= 1, phase= 0 id= 58
I'm Faulty
setting id e val of rap data message equal to59and0
sending rap message with val=0
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

Il protocollo ha termine anche questa volta con successo: $\boldsymbol{Tf = 17.8\ secondi}$

```
** Event #944.  T=17.376234  (17.37s).  Module #58 `net80211.host[3].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:59 val:0
aircraftID:31 val:1
aircraftID:73 val:1
aircraftID:45 val:1
aircraftID:17 val:1
---
 numoccurence= 1 i = 0
 numoccurence= 4 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #945.  T=17.441797  (17.44s).  Module #16 `net80211.host[0].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:17 val:1
aircraftID:31 val:1
aircraftID:59 val:0
aircraftID:73 val:1
aircraftID:45 val:1
---
 numoccurence= 1 i = 0
 numoccurence= 4 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #947.  T=17.522497  (17.52s).  Module #72 `net80211.host[4].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:73 val:1
aircraftID:31 val:1
aircraftID:59 val:0
aircraftID:45 val:1
aircraftID:17 val:1
---
 numoccurence= 1 i = 0
 numoccurence= 4 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #949.  T=17.732524  (17.73s).  Module #30 `net80211.host[1].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:31 val:1
aircraftID:59 val:0
aircraftID:73 val:1
aircraftID:45 val:1
aircraftID:17 val:1
---
 numoccurence= 1 i = 0
 numoccurence= 4 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #953.  T=17.819762  (17.81s).  Module #44 `net80211.host[2].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:45 val:1
aircraftID:31 val:1
aircraftID:59 val:0
aircraftID:73 val:1
aircraftID:17 val:1
---
 numoccurence= 1 i = 0
 numoccurence= 4 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

Il tempo di esecuzione totale del protocollo con 4 host è pertanto:

$$\boldsymbol{Tempo\ di\ esecuzione\ totale} = \boldsymbol{Tf} - \boldsymbol{Ti} = 17.8 - 16 = \mathbf{1.8}\ \text{secondi.}$$

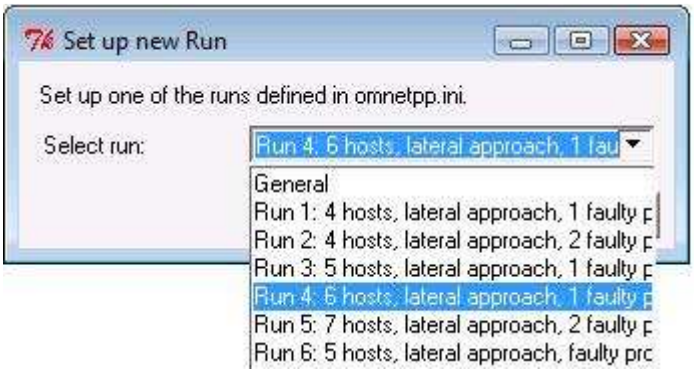
Il numero totale di pacchetti RAP, che possiamo ottenere consultando il file di output dove abbiamo scelto di registrare i messaggi RAP scambiati, ovvero il volume di traffico del protocollo è di 16 per ogni host, per un totale di 48.

$$\boldsymbol{Numero\ totale\ di\ messaggi\ RAP\ scambiati} = 16 * 5 = \mathbf{80}$$

7.3.3 Scenario 2: 6 host, 1 processo faulty.

M=1 e *N=6*: la condizione $\boldsymbol{N >= 3M + 1}$, è soddisfatta. Manteniamo inoltre le impostazioni di base:

- Il protocollo di agreement avverrà sulla coordinata **X** trasmessa dagli aircraft nel messaggio ADSB assieme a tutte le altre. Trattandosi di algoritmo binario è stata usata la convenzione che se la coordinata x comunicata è maggiore di 100 pixel, allora x_i=1, altrimenti x_i=0.
- Per quanto riguarda il processo faulty, è stato implementato semplicemente facendo in modo che invii un valore random per x_i, indipendentemente dalle informazioni che riceve dagli altri aircraft durante il protocollo. Non segue correttamente il protocollo.



Avviamo la simulazione.

$\boldsymbol{Ti = 16\ secondi}$, evento #170

```
** Event #169.  T=     16  (16.00s).  Module #13 `net80211.host[0].adsb'
handling message ADSB_TIMER
sending new adsb message
** Event #170.  T=     16  (16.00s).  Module #21 `net80211.host[0].wireless.sendSplitter'
sendersplitter: sended ADSBDataMessage message
** Event #171.  T=     16  (16.00s).  Module #17 `net80211.host[0].rap.logic'
RAPLogic: received agreement_start message
starting agreement with xi= 1, phase= 0 id= 17
setting id e val of rap data message equal to18and1
sending rap message with val=1
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

```
** Event #193.  T=16.001186  (16.00s).  Module #31 `net80211.host[1].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 15: 156,200
(RAPLogic) received adsb-in position data from 15 pos:(156,200)
received destination from aircraft 15: 400,200
(RAPLogic) received adsb-in destination data from 15 dst:(400,200)
received velocity from aircraft 15: 1
(RAPLogic) received adsb-in velocity data from 15 vel:1
received direction from aircraft 15: 0
(RAPLogic) received adsb-in direction data from 15 dir:1
 position of aircraft num 15is 156
starting agreement with xi= 1, phase= 0 id= 31
setting id e val of rap data message equal to32and1
sending rap message with val=1
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

L' Host #3 è il processo faulty:

```
** Event #202.  T=16.001186  (16.00s).  Module #59 `net80211.host[3].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 15: 156,200
(RAPLogic) received adsb-in position data from 15 pos:(156,200)
received destination from aircraft 15: 400,200
(RAPLogic) received adsb-in destination data from 15 dst:(400,200)
received velocity from aircraft 15: 1
(RAPLogic) received adsb-in velocity data from 15 vel:1
received direction from aircraft 15: 0
(RAPLogic) received adsb-in direction data from 15 dir:1
 position of aircraft num 15is 156
starting agreement with xi= 1, phase= 0 id= 59
I'm Faulty
setting id e val of rap data message equal to60and0
sending rap message with val=0
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout

** Event #211.  T=16.001186  (16.00s).  Module #45 `net80211.host[2].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 15: 156,200
(RAPLogic) received adsb-in position data from 15 pos:(156,200)
received destination from aircraft 15: 400,200
(RAPLogic) received adsb-in destination data from 15 dst:(400,200)
received velocity from aircraft 15: 1
(RAPLogic) received adsb-in velocity data from 15 vel:1
received direction from aircraft 15: 0
(RAPLogic) received adsb-in direction data from 15 dir:1
 position of aircraft num 15is 156
starting agreement with xi= 1, phase= 0 id= 45
setting id e val of rap data message equal to46and1
sending rap message with val=1
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

```
** Event #220.  T=16.001187  (16.00s).  Module #87 `net80211.host[5].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 15: 156,200
(RAPLogic) received adsb-in position data from 15 pos:(156,200)
received destination from aircraft 15: 400,200
(RAPLogic) received adsb-in destination data from 15 dst:(400,200)
received velocity from aircraft 15: 1
(RAPLogic) received adsb-in velocity data from 15 vel:1
received direction from aircraft 15: 0
(RAPLogic) received adsb-in direction data from 15 dir:1
 position of aircraft num 15is 156
starting agreement with xi= 1, phase= 0 id= 87
setting id e val of rap data message equal to88and1
sending rap message with val=1
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

```
** Event #229.  T=16.001187  (16.00s).  Module #73 `net80211.host[4].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 15: 156,200
(RAPLogic) received adsb-in position data from 15 pos:(156,200)
received destination from aircraft 15: 400,200
(RAPLogic) received adsb-in destination data from 15 dst:(400,200)
received velocity from aircraft 15: 1
(RAPLogic) received adsb-in velocity data from 15 vel:1
received direction from aircraft 15: 0
(RAPLogic) received adsb-in direction data from 15 dir:1
 position of aircraft num 15is 156
starting agreement with xi= 1, phase= 0 id= 73
setting id e val of rap data message equal to74and1
sending rap message with val=1
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

Il protocollo ha termine a partire dall' **evento #1344,** quando gli host raggiungono correttamente l' agreement sul valore 1, il valore corretto che ci aspettiamo. Ecco le RAPTable:

```
** Event #1309.  T=17.531126  (17.53s).  Module #73 `net80211.host[4].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:74 val:1
aircraftID:88 val:1
aircraftID:32 val:1
aircraftID:18 val:1
aircraftID:60 val:0
aircraftID:46 val:1
---
 numoccurence= 1 i = 0
 numoccurence= 5 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #1310.  T=17.556926  (17.55s).  Module #45 `net80211.host[2].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:46 val:1
aircraftID:88 val:1
aircraftID:32 val:1
aircraftID:18 val:1
aircraftID:74 val:1
aircraftID:60 val:0
---
 numoccurence= 1 i = 0
 numoccurence= 5 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #1311.  T=17.568692  (17.56s).  Module #87 `net80211.host[5].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:88 val:1
aircraftID:32 val:1
aircraftID:18 val:1
aircraftID:74 val:1
aircraftID:60 val:0
aircraftID:46 val:1
---
 numoccurence= 1 i = 0
 numoccurence= 5 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #1314.  T=17.737267  (17.73s).  Module #17 `net80211.host[0].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:18 val:1
aircraftID:88 val:1
aircraftID:32 val:1
aircraftID:74 val:1
aircraftID:60 val:0
aircraftID:46 val:1
---
 numoccurence= 1 i = 0
 numoccurence= 5 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #1315.  T=17.766708  (17.76s).  Module #31 `net80211.host[1].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:32 val:1
aircraftID:88 val:1
aircraftID:18 val:1
aircraftID:74 val:1
aircraftID:60 val:0
aircraftID:46 val:1
---
 numoccurence= 1 i = 0
 numoccurence= 5 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #1317.  T=17.883002  (17.88s).  Module #59 `net80211.host[3].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:60 val:0
aircraftID:88 val:1
aircraftID:32 val:1
aircraftID:18 val:1
aircraftID:74 val:1
aircraftID:46 val:1
---
 numoccurence= 1 i = 0
 numoccurence= 5 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

$\boldsymbol{Tf} = 17.88\ secondi$, evento #1317**.**

$\boldsymbol{Tempo\ di\ esecuzione\ totale} = \boldsymbol{Tf} - \boldsymbol{Ti} = 17.88 - 16 = \mathbf{1.88}$ secondi.

Il numero totale di pacchetti RAP, estratto dal file di log dove registriamo il totale dei messaggi RAP scambiati, ovvero il volume di traffico del protocollo, è di 20 per ogni host, per un totale di 120.

$$\boldsymbol{Numero\ totale\ di\ messaggi\ RAP\ scambiati} = 20 * 6 = \mathbf{120}$$

7.3.1 Scenario 1: 7 host, 2 processi faulty.

Infine, con uno scenario con *N=7* host è possibile eseguire una simulazione con *M=2* processi faulty, mantenendo comunque valida la condizione $\boldsymbol{N >= 3M + 1}$.

Avviamo la simulazione, al solito l' host 0 sarà la nostra sorgente che darà il via al trigger del protocollo:

```
** Event #199.  T=     16  (16.00s).  Module #14 `net80211.host[0].adsb'
handling message ADSB_TIMER
sending new adsb message
** Event #200.  T=     16  (16.00s).  Module #22 `net80211.host[0].wireless.sendSplitter'
sendersplitter: sended ADSBDataMessage message
** Event #201.  T=     16  (16.00s).  Module #18 `net80211.host[0].rap.logic'
RAPLogic: received agreement_start message
starting agreement with xi= 1, phase= 0 id= 18
setting id e val of rap data message equal to19and1
sending rap message with val=1
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

Gli altri processi ricevono il messaggio di trigger e avviano anche loro l'esecuzione del protocollo:

```
** Event #224.  T=16.001186  (16.00s).  Module #32 `net80211.host[1].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 16: 156,200
(RAPLogic) received adsb-in position data from 16 pos:(156,200)
received destination from aircraft 16: 400,200
(RAPLogic) received adsb-in destination data from 16 dst:(400,200)
received velocity from aircraft 16: 1
(RAPLogic) received adsb-in velocity data from 16 vel:1
received direction from aircraft 16: 0
(RAPLogic) received adsb-in direction data from 16 dir:1
 position of aircraft num 16is 156
starting agreement with xi= 1, phase= 0 id= 32
setting id e val of rap data message equal to33and1
sending rap message with val=1
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
** Event #225.  T=16.001186  (16.00s).  Module #11 `net80211.rapCoordinator'
sync
updating counter in RapCoordinator 2
```

```
** Event #233.  T=16.001186  (16.00s).  Module #60 `net80211.host[3].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 16: 156,200
(RAPLogic) received adsb-in position data from 16 pos:(156,200)
received destination from aircraft 16: 400,200
(RAPLogic) received adsb-in destination data from 16 dst:(400,200)
received velocity from aircraft 16: 1
(RAPLogic) received adsb-in velocity data from 16 vel:1
received direction from aircraft 16: 0
(RAPLogic) received adsb-in direction data from 16 dir:1
 position of aircraft num 16is 156
starting agreement with xi= 1, phase= 0 id= 60
I'm Faulty
setting id e val of rap data message equal to61and0
sending rap message with val=0
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

```
** Event #242.  T=16.001186  (16.00s).  Module #46 `net80211.host[2].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 16: 156,200
(RAPLogic) received adsb-in position data from 16 pos:(156,200)
received destination from aircraft 16: 400,200
(RAPLogic) received adsb-in destination data from 16 dst:(400,200)
received velocity from aircraft 16: 1
(RAPLogic) received adsb-in velocity data from 16 vel:1
received direction from aircraft 16: 0
(RAPLogic) received adsb-in direction data from 16 dir:1
 position of aircraft num 16is 156
starting agreement with xi= 1, phase= 0 id= 46
setting id e val of rap data message equal to47and1
sending rap message with val=1
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

```
** Event #251.  T=16.001186  (16.00s).  Module #74 `net80211.host[4].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 16: 156,200
(RAPLogic) received adsb-in position data from 16 pos:(156,200)
received destination from aircraft 16: 400,200
(RAPLogic) received adsb-in destination data from 16 dst:(400,200)
received velocity from aircraft 16: 1
(RAPLogic) received adsb-in velocity data from 16 vel:1
received direction from aircraft 16: 0
(RAPLogic) received adsb-in direction data from 16 dir:1
 position of aircraft num 16is 156
starting agreement with xi= 1, phase= 0 id= 74
I'm Faulty
setting id e val of rap data message equal to75and0
sending rap message with val=0
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

```
** Event #260.  T=16.001186  (16.00s).  Module #88 `net80211.host[5].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 16: 156,200
(RAPLogic) received adsb-in position data from 16 pos:(156,200)
received destination from aircraft 16: 400,200
(RAPLogic) received adsb-in destination data from 16 dst:(400,200)
received velocity from aircraft 16: 1
(RAPLogic) received adsb-in velocity data from 16 vel:1
received direction from aircraft 16: 0
(RAPLogic) received adsb-in direction data from 16 dir:1
 position of aircraft num 16is 156
starting agreement with xi= 1, phase= 0 id= 88
setting id e val of rap data message equal to89and1
sending rap message with val=1
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

```
** Event #269.  T=16.001187  (16.00s).  Module #102 `net80211.host[6].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 16: 156,200
(RAPLogic) received adsb-in position data from 16 pos:(156,200)
received destination from aircraft 16: 400,200
(RAPLogic) received adsb-in destination data from 16 dst:(400,200)
received velocity from aircraft 16: 1
(RAPLogic) received adsb-in velocity data from 16 vel:1
received direction from aircraft 16: 0
(RAPLogic) received adsb-in direction data from 16 dir:1
 position of aircraft num 16is 156
starting agreement with xi= 1, phase= 0 id= 102
setting id e val of rap data message equal to103and1
sending rap message with val=1
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

I processi faulty sono, come vediamo dal log, l' host 3 e l host 4. Immediatamente dopo, ora che tutti i processi hanno eseguito la fase di "`agreement starting`" il `rap-coordinator` può inviare a tutti la `signal` di sincronizzazione:

```
** Event #270.  T=16.001187  (16.00s).  Module #11 `net80211.rapCoordinator'
sync
updating counter in RapCoordinator 7
RAP Coordinator is sending signal to all host, delay = 0.32769
RAP Coordinator is sending signal to all host, delay = 0.14537
RAP Coordinator is sending signal to all host, delay = 0.466612
RAP Coordinator is sending signal to all host, delay = 0.297348
RAP Coordinator is sending signal to all host, delay = 0.224243
RAP Coordinator is sending signal to all host, delay = 0.0646714
RAP Coordinator is sending signal to all host, delay = 0.103695
```

Termine della simulazione e risultati

Mostriamo le RAPTABLE degli aircraft dopo l'esecuzione dello step finale. Il protocollo all'istante:

$$\boldsymbol{Tf} \approx 17.89\ secondi.$$

```
** Event #1722.  T=17.566664  (17.56s).  Module #18 `net80211.host[0].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:19 val:1
aircraftID:47 val:1
aircraftID:75 val:0
aircraftID:103 val:1
aircraftID:33 val:1
aircraftID:89 val:1
aircraftID:61 val:0
---
 numoccurence= 2 i = 0
 numoccurence= 5 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #1724.  T=17.633005  (17.63s).  Module #74 `net80211.host[4].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:75 val:0
aircraftID:47 val:1
aircraftID:19 val:1
aircraftID:103 val:1
aircraftID:33 val:1
aircraftID:89 val:1
aircraftID:61 val:0
---
 numoccurence= 2 i = 0
 numoccurence= 5 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #1725.  T=17.678077  (17.67s).  Module #60 `net80211.host[3].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:61 val:0
aircraftID:47 val:1
aircraftID:75 val:0
aircraftID:19 val:1
aircraftID:103 val:1
aircraftID:33 val:1
aircraftID:89 val:1
---
 numoccurence= 2 i = 0
 numoccurence= 5 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1


** Event #1726.  T=17.695316  (17.69s).  Module #88 `net80211.host[5].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:89 val:1
aircraftID:47 val:1
aircraftID:75 val:0
aircraftID:19 val:1
aircraftID:103 val:1
aircraftID:33 val:1
aircraftID:61 val:0
---
 numoccurence= 2 i = 0
 numoccurence= 5 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1


** Event #1727.  T=17.790352  (17.79s).  Module #102 `net80211.host[6].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:103 val:1
aircraftID:47 val:1
aircraftID:75 val:0
aircraftID:19 val:1
aircraftID:33 val:1
aircraftID:89 val:1
aircraftID:61 val:0
---
 numoccurence= 2 i = 0
 numoccurence= 5 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #1731.  T=17.836798  (17.83s).  Module #46 `net80211.host[2].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:47 val:1
aircraftID:75 val:0
aircraftID:19 val:1
aircraftID:103 val:1
aircraftID:33 val:1
aircraftID:89 val:1
aircraftID:61 val:0
---
 numoccurence= 2 i = 0
 numoccurence= 5 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

```
** Event #1732.  T=17.891191  (17.89s).  Module #32 `net80211.host[1].rap.logic'
RAPLogic: received signal message
 signal arrived
agreement phase 1
agreement round 3
executing switch case 1
deliver
RAP TABLE:
aircraftID:33 val:1
aircraftID:47 val:1
aircraftID:75 val:0
aircraftID:19 val:1
aircraftID:103 val:1
aircraftID:89 val:1
aircraftID:61 val:0
---
 numoccurence= 2 i = 0
 numoccurence= 5 i = 1
 numoccurence maggiore di 2f
 phase = 1 stop= 1
next round is4
 AGREEMENT REACHED IS 1
```

$$\boldsymbol{Tempo\ di\ esecuzione\ totale} = \boldsymbol{Tf} - \boldsymbol{Ti} = 17.89 - 16 = \mathbf{1.89}$$

Il numero di messaggi RAP totale scambiato è come già ci aspettavamo pari a:

$$\boldsymbol{Numero\ totale\ di\ messaggi\ RAP\ scambiati} = 7 * 24 = \mathbf{168}$$

Il numero dei messaggi scambiati all' interno di un istanza del protocollo RAPTOR è sempre pari a

$$\boldsymbol{Totale\ messaggi\ RAP\ scambiati} = 4N * (N - 1)$$

Si tratta di un algoritmo la cui complessità in termini di numero di messaggi scambiati è dell' ordine $O(N^2)$, dove N è il numero degli host (scala col quadrato del numero degli host).

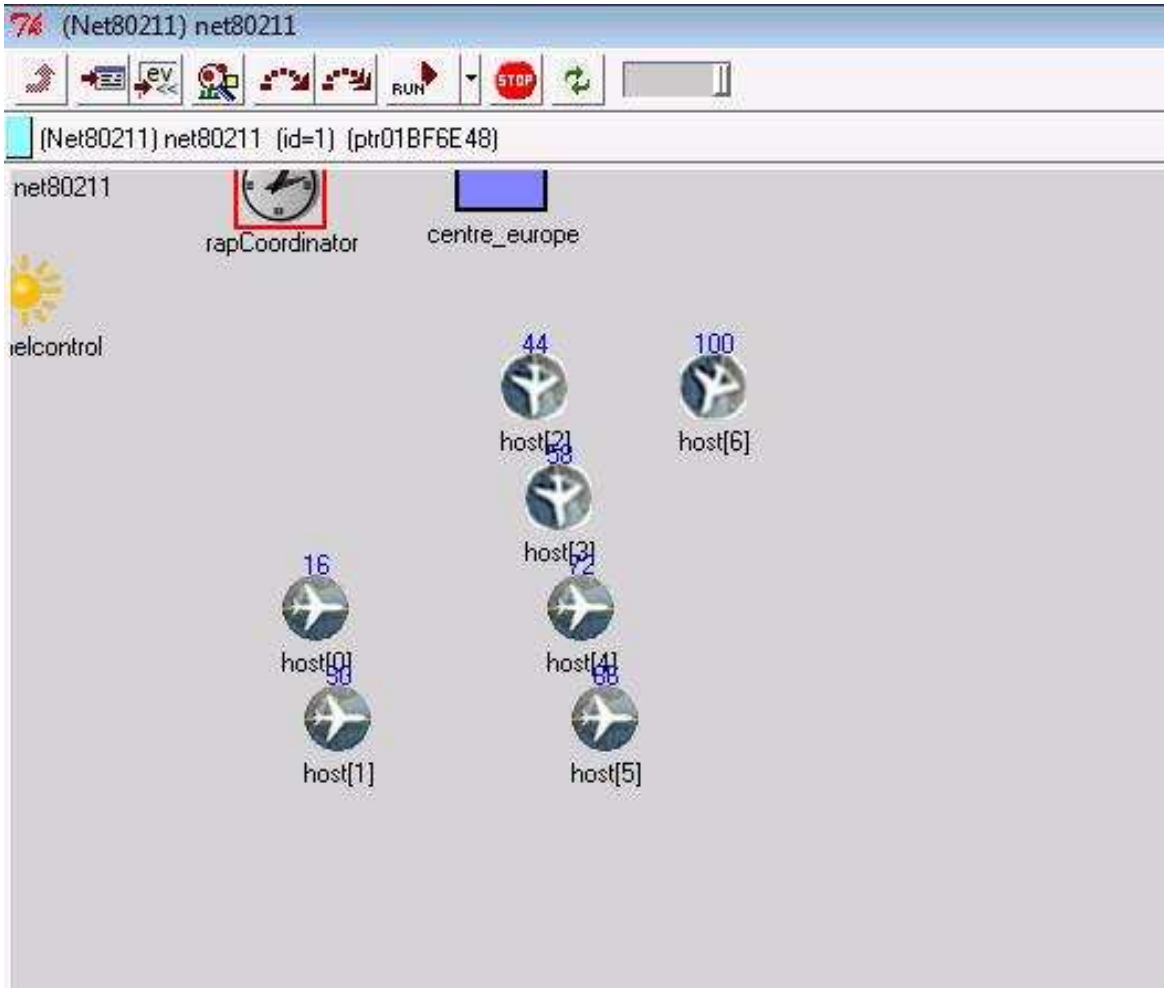
7.4 Simulazione algoritmo RAPTOR MULTIVALUED CONSENSUS

Eseguiamo ora il codice con cui abbiamo implementato l' algoritmo RAPTOR MULTIVALUED per l' agreement tra più processi su valori che appartengono ad un dato arbitrario dominio D.

Avendo visto nel dettaglio come funziona l' algoritmo binario una volta eseguito, analizzeremo da subito una simulazione più complessa, dove sono presenti 7 aircraft di cui 2 faulty scelti arbitrariamente dall' utente all' avvio della simulazione. Inseriamo dapprima il numero di processi faulty presenti nel sistema:

In seguito comunichiamo all'applicazione sottoforma di lista di id, quali sono gli host che desideriamo agiscano da processi faulty all' interno del sistema, nell' esempio, l' aircraft 4 e l' aircraft 6:

Avviamo la simulazione, lo scenario iniziale è riportato in figura:



L' istante di trigger del protocollo da parte del processo sorgente **0** è $\boldsymbol{Ti} = 16\ secondi$:

```
** Event #201.  T=     16  (16.00s).  Module #18 `net80211.host[0].rap.logic'
RAPLogic: received agreement_start message
STARTING MULTIVALUED agreement with xi= 156, phase= 0 id= 18
setting id e val of rap data message equal to19and156
sending rap message with val=156
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

L' agreement dovrà avvenire sul valore di coordinata x=156. I 2 processi faulty sono codificati per spedire messaggi RAP contenenti valori random. Il processo faulty 4 inizia il protocollo inviando x=75 pur avendo ricevuto x=156:

```
** Event #251.  T=16.001186  (16.00s).  Module #74 `net80211.host[4].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 16: 156,200
(RAPLogic) received adsb-in position data from 16 pos:(156,200)
received destination from aircraft 16: 400,200
(RAPLogic) received adsb-in destination data from 16 dst:(400,200)
received velocity from aircraft 16: 1
(RAPLogic) received adsb-in velocity data from 16 vel:1
received direction from aircraft 16: 0
(RAPLogic) received adsb-in direction data from 16 dir:1
 position of aircraft num 16is 156
STARTING MULTIVALUED agreement with xi= 156, phase= 0 id= 74
I'm Faulty
setting id e val of rap data message equal to75and41
sending rap message with val=41
success broadcast
calling sync
```

Il processo faulty 6 inizia il protocollo inviando invece x=75

```
** Event #269.  T=16.001187  (16.00s).  Module #102 `net80211.host[6].rap.logic'
RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 16: 156,200
(RAPLogic) received adsb-in position data from 16 pos:(156,200)
received destination from aircraft 16: 400,200
(RAPLogic) received adsb-in destination data from 16 dst:(400,200)
received velocity from aircraft 16: 1
(RAPLogic) received adsb-in velocity data from 16 vel:1
received direction from aircraft 16: 0
(RAPLogic) received adsb-in direction data from 16 dir:1
 position of aircraft num 16is 156
STARTING MULTIVALUED agreement with xi= 156, phase= 0 id= 102
I'm Faulty
setting id e val of rap data message equal to103and67
sending rap message with val=67
success broadcast
calling sync
sendig sync mess to rapcoordinator via syncout
```

Al termine della prima parte dell' algoritmo multivalued e dei primi due cicli di scambio di messaggi-RAP, dando un occhiata al resoconto della simulazione, le RAPTABLE su ogni aircraft risultano essere le seguenti:

```
** Event #1033.  T=17.468269  (17.46s).  Module #18 `net80211.host[0].rap.logic'
RAPLogic: received signal message
 signal arrived
multivalued_flag is true
multivalued agreement phase 0, multivalued agreement round 2
multivalued agreement: executing switch case 2, delivering...
deliver
RAP TABLE:
aircraftID:19 val:156
aircraftID:47 val:156
aircraftID:75 val:334
aircraftID:103 val:100
aircraftID:33 val:156
aircraftID:89 val:156
aircraftID:61 val:156
```

```
** Event #1019.  T=16.929392  (16.92s).  Module #32 `net80211.host[1].rap.logic'
RAPLogic: received signal message
 signal arrived
multivalued_flag is true
multivalued agreement phase 0, multivalued agreement round 2
multivalued agreement: executing switch case 2, delivering...
deliver
RAP TABLE:
aircraftID:33 val:156
aircraftID:47 val:156
aircraftID:75 val:334
aircraftID:19 val:156
aircraftID:103 val:100
aircraftID:89 val:156
aircraftID:61 val:156
```

```
** Event #1026.  T=17.151624  (17.15s).  Module #46 `net80211.host[2].rap.logic'
RAPLogic: received signal message
 signal arrived
multivalued_flag is true
multivalued agreement phase 0, multivalued agreement round 2
multivalued agreement: executing switch case 2, delivering...
deliver
RAP TABLE:
aircraftID:47 val:156
aircraftID:75 val:334
aircraftID:19 val:156
aircraftID:103 val:100
aircraftID:33 val:156
aircraftID:89 val:156
aircraftID:61 val:156
```

```
** Event #1016.  T=16.855733  (16.85s).  Module #60 `net80211.host[3].rap.logic'
RAPLogic: received signal message
 signal arrived
multivalued_flag is true
multivalued agreement phase 0, multivalued agreement round 2
multivalued agreement: executing switch case 2, delivering...
deliver
RAP TABLE:
aircraftID:61 val:156
aircraftID:47 val:156
aircraftID:75 val:334
aircraftID:19 val:156
aircraftID:103 val:100
aircraftID:33 val:156
aircraftID:89 val:156
```

```
** Event #1021.  T=16.93982  (16.93s).  Module #74 `net80211.host[4].rap.logic'
RAPLogic: received signal message
 signal arrived
multivalued_flag is true
multivalued agreement phase 0, multivalued agreement round 2
multivalued agreement: executing switch case 2, delivering...
deliver
RAP TABLE:
aircraftID:75 val:334
aircraftID:47 val:156
aircraftID:19 val:156
aircraftID:103 val:100
aircraftID:33 val:156
aircraftID:89 val:156
aircraftID:61 val:156
```

```
** Event #1030.  T=17.270665  (17.27s).  Module #88 `net80211.host[5].rap.logic'
RAPLogic: received signal message
 signal arrived
multivalued_flag is true
multivalued agreement phase 0, multivalued agreement round 2
multivalued agreement: executing switch case 2, delivering...
deliver
RAP TABLE:
aircraftID:89 val:156
aircraftID:47 val:156
aircraftID:75 val:334
aircraftID:19 val:156
aircraftID:103 val:100
aircraftID:33 val:156
aircraftID:61 val:156
```

```
** Event #1028.  T=17.163416  (17.16s).  Module #102 `net80211.host[6].rap.logic'
RAPLogic: received signal message
 signal arrived
multivalued_flag is true
multivalued agreement phase 0, multivalued agreement round 2
multivalued agreement: executing switch case 2, delivering...
deliver
RAP TABLE:
aircraftID:103 val:100
aircraftID:47 val:156
aircraftID:75 val:334
aircraftID:19 val:156
aircraftID:33 val:156
aircraftID:89 val:156
aircraftID:61 val:156
```

Ogni processo memorizza il valore, se esiste, all' interno della RAPTable tale per cui la condizione:

$$\textbf{if } \exists_{v \neq \perp} : \#_v(V_i) \geq 2f + 1$$

è soddisfatta, nel nostro caso *v*=156.

Inizia a questo punto, come abbiamo visto nel precedente, l' algoritmo RAPTOR binario per vedere se è possibile un accordo. Il tempo di esecuzione simulato impiegato da questa prima parte del protocollo dovuta ad un dominio di tipo non binario è pari a

$$\boldsymbol{Tf - Ti} = 17.163416 - 16.00 \approx 1.16\ secondi.$$

A questo punto ha inizio l' agreement binario per determinare se c'è un valore tra quelli proposti su cui è possibile accordarsi (risultato dell' algoritmo binario uguale a 1) oppure no (l'agreement binario ritorna 0). Nel secondo caso i processi decideranno su un valore di default stabilito a priori, nel nostro codice pari a 100: `#define d_default 100.`

Nel seguito mostriamo un estratto dal file di log che riporta i dati relativi ai tempi di simulazione e al volume di traffico

File Modifica Formato Visualizza ?

```
run 5 "net80211"
scalar "net80211.host[0].rap.logic"     "Total Execution Time"  4.70690631774
scalar "net80211.host[0].rap.comm"      "Num of rap message received"   36

scalar "net80211.host[1].rap.logic"     "Total Execution Time"  4.70210163185
scalar "net80211.host[1].rap.comm"      "Num of rap message received"   36

scalar "net80211.host[2].rap.logic"     "Total Execution Time"  4.68899682654
scalar "net80211.host[2].rap.comm"      "Num of rap message received"   36

scalar "net80211.host[3].rap.logic"     "Total Execution Time"  4.91616076506
scalar "net80211.host[3].rap.comm"      "Num of rap message received"   36

scalar "net80211.host[4].rap.logic"     "Total Execution Time"  4.87423650842
scalar "net80211.host[4].rap.comm"      "Num of rap message received"   36

scalar "net80211.host[5].rap.logic"     "Total Execution Time"  4.90641042183
scalar "net80211.host[5].rap.comm"      "Num of rap message received"   36

scalar "net80211.host[6].rap.logic"     "Total Execution Time"  4.8364858333
scalar "net80211.host[6].rap.comm"      "Num of rap message received"   36
```

Il tempo medio impiegato da ogni processo a completare il protocollo è pari quindi a:

$$\boldsymbol{Tempo\ di\ esecuzione\ medio} \cong 4.804\ secondi$$

Il numero totale di pacchetti RAP scambiati, e quindi il volume di traffico del protocollo RAP multivalued è pari a 36 pacchetti per ogni aircraft:

$$\boldsymbol{Numero\ totale\ di\ messaggi\ RAP\ scambiati} = 36 * 7 = 252$$

Rispetto al semplice protocollo RAP binario, nelle stesse condizioni di sistema (7 aircraft, 2 faulty) possiamo notare un incremento significativo del 50% del volume di traffico, e del 100% per quanto riguarda il tempo di esecuzione totale.

CAPITOLO 8

IMPLEMENTAZIONE DELL'ALGORITMO DEI GENERALI BIZANTINI IN OMNeT++

All' interno della nostra applicazione i vari moduli RAPLogic presenti sugli Aircraft, rappresentano i processi che devono accordarsi sullo stesso valore. L' algoritmo dei Generali Bizantini, le sue strutture dati, e l' algoritmo di decisione vengono implementati pertanto in questi moduli.

8.1 Le strutture dati

La parte privata del modulo RAPLogic si presenta così:

```
class RAPLogic : public cSimpleModule {
   private:
      RAPComm* comm_ptr;
      DataTable dt;
      Node xi;
      int round;
      enum phase {delivering, sending};
```

```
      phase p;
      int mId;                      //The ID of the process
      std::map<Path,Node> mNodes;   //The map that holds the process tree


      // Static data shared among all RAPLogic Modules
      static Traits mTraits;
      static std::map<Path, std::vector<Path> > mChildren;
            static      std::map<size_t,std::map<size_t,std::vector<Path>>>
mPathsByRank;
      ....
      ....
      ....
      }
```

Vengono aggiunti 3 membri statici, condivisi da tutti i processi.

-static Process mProcess:

è semplicemente una classe di supporto che definisce il comportamento dei processi (faulty o meno),

il tie-breaker utilizzato oltre ad alcune routine di utility frequentemente utilizzate.

```
class Process {
public :
    Process( int source, int m, int n, bool debug = false );
    Node GetSourceValue();
    // tie breaker
    char GetDefault();
    bool IsFaulty( int process );
    int mSource;
    int GetSource;
    void SetSource (int id);
    // These two members hold M and N, the number of rounds and
    // the number of processes.
```

```
    int mM;
    int mN;
    const bool mDebug;
};
```

Modificando questa classe in modo oppurtuno è possibile modificare il comportamento dei singoli processi, il tie breaker utilizzato etc…

-static std::map<Path, std::vector<Path> > mChildren:

Questa mappa viene popolata inizialmente dal primo processo RAPLogic che inizia l' esecuzione dell' algoritmo di Lamport dopo aver ricevuto un messaggio ADSB dal processo Source, e contiene una copia statica dell' albero. Capita molte volte nel corso dell' esecuzione dell' algoritmo che dato un path vogliamo conoscere tutti i figli per quel dato nodo: `mChildren[Path]`, semplicemente ritorna un vettore con tutti i figli per il Path dato.

-static std::map<size_t,std::map<size_t,std::vector<Path>> > mPathsByRank;

questa mappa contiene tutti i path per ciascun round di esecuzione dell' algoritmo e Aircraft ID. Da essa è possibile ricavare il numero di messaggi che un dato Aircraft deve brodcastare in un dato round, così come il path in cui un dato messaggio ricevuto in un dato round da un determinato AircraftID deve essere memorizzato all' interno dell' Information Tree.

8.2 Funzionamento

Il primo Aircraft che esegue il broadcast del messaggio ADSB contenente le proprie informazioni su velocità, posizione, accelerazione, assurge al ruolo di processo Source.

Per tanto una volta eseguito il broadcast del messaggio ADSB non partecipa più all' esecuzione dell' algoritmo: non invia più messaggi, né ne riceverà dagli altri aircraft (semplicemente vengono scartati). Tutti gli altri Aircraft, una volta ricevuto il messaggio ADSB, semplicemente ricavano il valore proposto dalla Sorgente (`char value`), e iniziano l' esecuzione dell' algoritmo di Lamport.

Inoltre, il primo tra loro che riceve il messaggio ADSB del processo Source, va anche ad inizializzare la struttura `mChildren.`

```
void RAPLogic::handleMessage(cMessage* msg_ptr) {
 ev<<endl<<"RAPLogic: received "<<msg_ptr->name()<<" message"<<endl;
 if (strcmp(msg_ptr->name(), "signal") == 0)  {
     if (IsSource()){ev<<"signal discarded, i'm the source”<<endl; return;}
     int ret=RunLamport();
  }
 else if(dynamic_cast<ADSBDataMessage*>(msg_ptr) != NULL) {
   //collects ADSB data. The agreement will be executed on these data
   storeData((ADSBDataMessage*)msg_ptr);
   double position=(dynamic_cast<ADSBDataMessage*>(msg_ptr))
                                                    ->getPosition().x;
   Char value=(((dynamic_cast<ADSBDataMessage*>(msg_ptr))
                                          ->getPosition()).x)>100? One:Zero;
   int Aircraft_ID=(dynamic_cast<ADSBDataMessage*>(msg_ptr))
                                                    ->getAircraftID();
   int SourceHost=((Aircraft_ID-16)/14);
   SetSource(SourceHost);
   ev<<"setting source host "<<SourceHost<<endl;
   if ( mChildren.size() == 0 ) {
       GenerateChildren(mProcess.mM,mProcess.mN,
                          std::vector<bool>(Process.mN,true,SourceHost);

       xi.input_value= value;
```

```
        ReceiveMessage(mPathsByRank[0][SourceHost][0],Node(value,UNKNOWN));
        RunLamport();
    } }
```

Analizziamo quindi ora la funzione Run_Lamport, che implementa l'algoritmo dei Generali Bizantini vero e proprio.La Run_Lamport si alterna ad ogni esecuzione tra due fasi: quella di *sending* (broadcast dei vari messaggi da parte di tutti gli Aircraft coinvolti) o quella di *delivering* (collezione dei messaggi spediti nella fase precedente), fino al raggiungimento del numero di round necessari per pervenire ad una corretta decisione (che sappiamo essere uguale al numero di processi faulty presenti).

La fase di *sending* è piuttosto semplice:

```
int RAPLogic::RunLamport()
{
 // Starting at round 0 and working up to round M, call the
 // SendMessages() method of each process. It will send the appropriate
 // message to all other processes.
 Node* val=0;
 int num_of_messages;
 switch (p) {
      case sending:
            if (round>M) {
            //Print out the results. For non-faulty processes,we call
            //the Decide() method
                  round=1;
                  if (!IsFaulty())
                    ev<<"Process"<<mId<<"decides on value"<<Decide()<<endl;
                  else {
                      ev<<"Process "<<mId<<" is faulty"<<endl;
```

```
                    ev<<"end lamport"<<endl<<endl;
                    return 0;
                    }
                }
            else {
                    ev<<"broadcasting messages of round"<<round<<endl;
                    SendMessages();
                }
                sync(N);
                p=delivering;
                break;
        case delivering:
                ..............
                ..............
                ..............
                }
return 0;
}
```

Se siamo arrivati al numero di round prestabilito, semplicemente la funzione chiama il metodo `Decide()` se il processo non è faulty, e ritorna.

Altrimenti chiama la `SendMessages()` che si occupa dei broadcast dei vari messaggi per il round attuale e si sincronizza , con la `sync(N),` in attesa così che anche gli altri Aircraft eseguano le loro broadcast per quel determinato round di esecuzione. Avviene lo switch su *delivering* e ritorna.

Non appena tutti gli Aircraft hanno inviato i loro messaggi in broadcast, e pertanto hanno eseguito tutti la `sync(N),` spedendo un messaggio di sincronizzazione al RAPCoordinator, quest' ultimo invia una signal in broadcast a tutti gli Aircraft presenti.

Alla ricezione della signal viene nuovamente richiamata la Run_Lamport che esegue la seguente parte di *delivering:*

```
int RAPLogic::RunLamport()
{
  Node* val=0;
  int num_of_messages;
  switch (p) {
     case sending:
        ......
        ......
     case delivering:
        p=sending;
        RAPTable rt=deliver();
      // from aircraftID to nodeID for correctly indexing mPathsByRank
        int Aircraft_node_Id;
        const int * idTab;
        idTab=rt.getAircraftIDs();
        for (int i=0; i<N-1;i++ ){
            int Aircraft_ID= (idTab[i]);
            rt.getNumMsgperId (Aircraft_ID,num_of_messages);
            //num of messages in RAPTable having a certain aircraft id
            val= new Node [num_of_messages];
            rt.getVal(Aircraft_ID, val);
            Aircraft_node_Id=((Aircraft_ID-16)/14);
            for (int j=0; j< num_of_messages; j++)
           ReceiveMessage(mPathsByRank[round][Aircraft_node_Id][j],val[j]);
            }
         sync(N);
         round++;
```

```
            break;
            }
        return 0;
}
```

Per prima la vengono memorizzati nel vettore di nodi *val,* tutti i messaggi dello stesso *Aircraft_ID* dopo di che, con il secondo *for,* su ogni nodo contenuto in *val* viene eseguita la *ReceiveMessage* che semplicemente memorizza il nodo nella struttura dati *mNodes[]* del modulo RAPLogic su cui poi andrà a lavorare l' algoritmo per prendere la sua decisione finale.

A questo si incrementa il round, ci si sincronizza nuovamente in attesa che tutti gli Aircraft eseguano la loro fase di delivering e la funziona ritorna.

Una volta che il numero di *round* ha raggiunto il valore M = numero di faulty process presenti nella simulazione, verrà chiamato come visto sopra il metodo `Decide()`*:*

```
char RAPLogic::Decide() {
   // Step 1 - set the leaf values
   ev<<"Taking a decision: beginning step one: set the leaf value"<<endl;
   for ( size_t i = 0 ; i < mTraits.mN ; i++ )
       for( size_t j = 0 ; j < mPathsByRank[ mTraits.mM ][ i ].size();j++){
                const Path &path = mPathsByRank[ mTraits.mM ][ i ][ j ];
                Node &node = mNodes[ path ];
                node.output_value = node.input_value;
            }
   // Step 2 - work up the tree
   ev<<"Taking a decision: beginning step two: work up the tree"<<endl;
   for ( int round = (int) mTraits.mM - 1 ; round >= 0 ; round-- )
        {
            for ( size_t i = 0 ; i < mTraits.mN ; i++ )
                for ( size_t j=0; j<mPathsByRank[ round ][ i ].size(); j++)
```

```
                {
                    const Path &path = mPathsByRank[ round ][ i ][ j ];
                    Node &node = mNodes[ path ];
                    node.output_value = GetMajority( path );
                        }
        }
    const Path &top_path = (mPathsByRank[ 0 ][ mTraits.mSource ]).front();
        const Node &top_node = mNodes[ top_path ];
        return top_node.output_value;
     }
```

Il metodo lavora esattamente come descritto in precedenza: un primo passo, pone il valore di *output* di ogni nodo foglia , uguale al proprio valore di *input* (i vari valori ricevuti dalle broadcast nei vari round dell' algoritmo), e un secondo passo scorre l' albero fino alla radice utilizzando la funzione *GetMajority()* di volta in volta per settare il valore di *output* di ogni nodo padre. Arrivati alla radice dell' albero, il metodo ritorna come valore di decisione , il valore di *output* della stessa, e l' applicazione termina.

8.3 Esempio di una simulazione dell' algoritmo implementato: 7 host, 2 processi faulty.

Processo Sorgente: Host 0.

Processi Faulty: Host 1, Host 3.

Il processo 0 invia il primo messaggio ADSB assumendo il ruolo di processo sorgente e non parteciperà più da ora in avanti all'esecuzione del protocollo. Gli Hosts 1 e 3 sono stati configurati appositamente come processi faulty.

Come abbiamo visto, nell protocollo di Lamport, quando gli aircraft ricevono il messaggio ADSB del processo sorgente, iniziano a ritrasmettere il messaggio, come mostrato di seguito:

Istante di inizio del protocollo Ti $=$ $16{,}00$ $secondi$.

```
** Event #219.  T=16.001186  (16.00s).  Module #32 `net80211.host[1].rap.logic'

RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 16: 156,200
(RAPLogic) received adsb-in position data from 16 pos:(156,200)
received destination from aircraft 16: 400,200
(RAPLogic) received adsb-in destination data from 16 dst:(400,200)
received velocity from aircraft 16: 1
(RAPLogic) received adsb-in velocity data from 16 vel:1
received direction from aircraft 16: 0
(RAPLogic) received adsb-in direction data from 16 dir:1
lamport case sending round number 1, M= 2
broadcasting messages of round1
broadcasting message...
broadcasting rap message
success broadcast
sending sync mess to rapcoordinator via syncout

** Event #227.  T=16.001186  (16.00s).  Module #60 `net80211.host[3].rap.logic'

RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 16: 156,200
(RAPLogic) received adsb-in position data from 16 pos:(156,200)
received destination from aircraft 16: 400,200
(RAPLogic) received adsb-in destination data from 16 dst:(400,200)
received velocity from aircraft 16: 1
(RAPLogic) received adsb-in velocity data from 16 vel:1
received direction from aircraft 16: 0
(RAPLogic) received adsb-in direction data from 16 dir:1
lamport case sending round number 1, M= 2
broadcasting messages of round1
broadcasting message...
broadcasting rap message
success broadcast
sending sync mess to rapcoordinator via syncout
```

```
** Event #235.  T=16.001186  (16.00s).  Module #46 `net80211.host[2].rap.logic'

RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 16: 156,200
(RAPLogic) received adsb-in position data from 16 pos:(156,200)
received destination from aircraft 16: 400,200
(RAPLogic) received adsb-in destination data from 16 dst:(400,200)
received velocity from aircraft 16: 1
(RAPLogic) received adsb-in velocity data from 16 vel:1
received direction from aircraft 16: 0
(RAPLogic) received adsb-in direction data from 16 dir:1
lamport case sending round number 1, M= 2
broadcasting messages of round1
broadcasting message...
broadcasting rap message
success broadcast
sending sync mess to rapcoordinator via syncout
```

```
** Event #243.  T=16.001186  (16.00s).  Module #74 `net80211.host[4].rap.logic'

RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 16: 156,200
(RAPLogic) received adsb-in position data from 16 pos:(156,200)
received destination from aircraft 16: 400,200
(RAPLogic) received adsb-in destination data from 16 dst:(400,200)
received velocity from aircraft 16: 1
(RAPLogic) received adsb-in velocity data from 16 vel:1
received direction from aircraft 16: 0
(RAPLogic) received adsb-in direction data from 16 dir:1
lamport case sending round number 1, M= 2
broadcasting messages of round1
broadcasting message...
broadcasting rap message
success broadcast
sending sync mess to rapcoordinator via syncout
```

```
** Event #251.  T=16.001186  (16.00s).  Module #88 `net80211.host[5].rap.logic'

RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 16: 156,200
(RAPLogic) received adsb-in position data from 16 pos:(156,200)
received destination from aircraft 16: 400,200
(RAPLogic) received adsb-in destination data from 16 dst:(400,200)
received velocity from aircraft 16: 1
(RAPLogic) received adsb-in velocity data from 16 vel:1
received direction from aircraft 16: 0
(RAPLogic) received adsb-in direction data from 16 dir:1
lamport case sending round number 1, M= 2
broadcasting messages of round1
broadcasting message...
broadcasting rap message
success broadcast
sending sync mess to rapcoordinator via syncout
```

```
** Event #259.  T=16.001187  (16.00s).  Module #102 `net80211.host[6].rap.logic'

RAPLogic: received ADSBDataMessage message
RAPLogic: storing ADSBDataMessage
received position from aircraft 16: 156,200
(RAPLogic) received adsb-in position data from 16 pos:(156,200)
received destination from aircraft 16: 400,200
(RAPLogic) received adsb-in destination data from 16 dst:(400,200)
received velocity from aircraft 16: 1
(RAPLogic) received adsb-in velocity data from 16 vel:1
received direction from aircraft 16: 0
(RAPLogic) received adsb-in direction data from 16 dir:1
lamport case sending round number 1, M= 2
broadcasting messages of round1
broadcasting message...
broadcasting rap message
success broadcast
sending sync mess to rapcoordinator via syncout
```

Durante l' iterazione 1 del protocollo il numero di messaggi inviati in totale è pari a *(N-1)=6,* come mostriamo da un estratto dei log di simulazione:

```
** Event #564.  T=16.093558  (16.09s).  Module #88 `net80211.host[5].rap.logic'

RAPLogic: received signal message
lamport case delivering round number 1
num of messages received from Aircraft 89 is 1
num of messages received from Aircraft 47 is 1
num of messages received from Aircraft 61 is 1
num of messages received from Aircraft 75 is 1
num of messages received from Aircraft 33 is 1
num of messages received from Aircraft 103 is 1
sending sync mess to rapcoordinator via syncout

** Event #581.  T=17.135429  (17.13s).  Module #32 `net80211.host[1].rap.logic'

RAPLogic: received signal message
lamport case delivering round number 1
num of messages received from Aircraft 33 is 1
num of messages received from Aircraft 47 is 1
num of messages received from Aircraft 61 is 1
num of messages received from Aircraft 89 is 1
num of messages received from Aircraft 75 is 1
num of messages received from Aircraft 103 is 1
sending sync mess to rapcoordinator via syncout

** Event #604.  T=19.107265  (19.10s).  Module #74 `net80211.host[4].rap.logic'

RAPLogic: received signal message
lamport case delivering round number 1
num of messages received from Aircraft 75 is 1
num of messages received from Aircraft 47 is 1
num of messages received from Aircraft 61 is 1
num of messages received from Aircraft 89 is 1
num of messages received from Aircraft 33 is 1
num of messages received from Aircraft 103 is 1
sending sync mess to rapcoordinator via syncout
```

```
** Event #621.  T=20.407064  (20.40s).  Module #102 `net80211.host[6].rap.logic'

RAPLogic: received signal message
lamport case delivering round number 1
num of messages received from Aircraft 103 is 1
num of messages received from Aircraft 47 is 1
num of messages received from Aircraft 61 is 1
num of messages received from Aircraft 89 is 1
num of messages received from Aircraft 75 is 1
num of messages received from Aircraft 33 is 1
sending sync mess to rapcoordinator via syncout

** Event #629.  T=20.934894  (20.93s).  Module #60 `net80211.host[3].rap.logic'

RAPLogic: received signal message
lamport case delivering round number 1
num of messages received from Aircraft 61 is 1
num of messages received from Aircraft 47 is 1
num of messages received from Aircraft 89 is 1
num of messages received from Aircraft 75 is 1
num of messages received from Aircraft 33 is 1
num of messages received from Aircraft 103 is 1
sending sync mess to rapcoordinator via syncout

** Event #681.  T=25.166483  (25.16s).  Module #46 `net80211.host[2].rap.logic'

RAPLogic: received signal message
lamport case delivering round number 1
num of messages received from Aircraft 47 is 1
num of messages received from Aircraft 61 is 1
num of messages received from Aircraft 89 is 1
num of messages received from Aircraft 75 is 1
num of messages received from Aircraft 33 is 1
num of messages received from Aircraft 103 is 1
sending sync mess to rapcoordinator via syncout
```

All' iterazione 2, ciascun host riceve un solo messaggi da ognuno degli altri host. All' iterazione 2 ci aspettiamo che il numero messaggi ricevuti scambiati sia:

$$(N-1\ ) * (N-2) = 6 * 5 = 30$$

```
** Event #2315.  T=36.053207  (36.05s).  Module #74 `net80211.host[4].rap.logic'

RAPLogic: received signal message
lamport case delivering round number 2
num of messages received from Aircraft 75 is 5
num of messages received from Aircraft 47 is 5
num of messages received from Aircraft 61 is 5
num of messages received from Aircraft 89 is 5
num of messages received from Aircraft 33 is 5
num of messages received from Aircraft 103 is 5
sending sync mess to rapcoordinator via syncout
```

```
** Event #2346.  T=38.508246  (38.50s).  Module #102 `net80211.host[6].rap.logic'

RAPLogic: received signal message
lamport case delivering round number 2
num of messages received from Aircraft 103 is 5
num of messages received from Aircraft 47 is 5
num of messages received from Aircraft 61 is 5
num of messages received from Aircraft 89 is 5
num of messages received from Aircraft 75 is 5
num of messages received from Aircraft 33 is 5
sending sync mess to rapcoordinator via syncout
```

```
** Event #2359.  T=39.599707  (39.59s).  Module #46 `net80211.host[2].rap.logic'

RAPLogic: received signal message
lamport case delivering round number 2
num of messages received from Aircraft 47 is 5
num of messages received from Aircraft 61 is 5
num of messages received from Aircraft 89 is 5
num of messages received from Aircraft 75 is 5
num of messages received from Aircraft 33 is 5
num of messages received from Aircraft 103 is 5
sending sync mess to rapcoordinator via syncout
```

```
** Event #2372.  T=40.103275  (40.10s).  Module #88 `net80211.host[5].rap.logic'

RAPLogic: received signal message
lamport case delivering round number 2
num of messages received from Aircraft 89 is 5
num of messages received from Aircraft 47 is 5
num of messages received from Aircraft 61 is 5
num of messages received from Aircraft 75 is 5
num of messages received from Aircraft 33 is 5
num of messages received from Aircraft 103 is 5
sending sync mess to rapcoordinator via syncout
```

```
** Event #2375.  T=40.154327  (40.15s).  Module #60 `net80211.host[3].rap.logic'

RAPLogic: received signal message
lamport case delivering round number 2
num of messages received from Aircraft 61 is 5
num of messages received from Aircraft 47 is 5
num of messages received from Aircraft 89 is 5
num of messages received from Aircraft 75 is 5
num of messages received from Aircraft 33 is 5
num of messages received from Aircraft 103 is 5
sending sync mess to rapcoordinator via syncout
```

```
** Event #2388.  T=41.328742  (41.32s).  Module #32 `net80211.host[1].rap.logic'

RAPLogic: received signal message
lamport case delivering round number 2
num of messages received from Aircraft 33 is 5
num of messages received from Aircraft 47 is 5
num of messages received from Aircraft 61 is 5
num of messages received from Aircraft 89 is 5
num of messages received from Aircraft 75 is 5
num of messages received from Aircraft 103 is 5
sending sync mess to rapcoordinator via syncout
```

Ogni host riceve 5 messaggi all' iterazione numero 2, per un totale di 180 messaggi totali già alla seconda iterazione. Andiamo ora a vedere i tempi di esecuzione del protocollo.

```
** Event #2390.  T=41.562866  (41.56s).  Module #46 `net80211.host[2].rap.logic'

RAPLogic: received signal message
lamport case sending round number 3, M= 2
Taking a decision: beginning step one: set the leaf value
Taking a decision: beginning step two: work up the tree
Taking a decision: beginning step one: set the leaf value
Taking a decision: beginning step two: work up the tree
Process 2 decides on value 1
end lamport, tf=41.5629 total execution time is 25.5617
```

```
** Event #2399.  T=41.983871  (41.98s).  Module #60 `net80211.host[3].rap.logic'

RAPLogic: received signal message
lamport case sending round number 3, M= 2
Process 3 is faulty
end lamport, tf=41.9839 total execution time is 25.9827
```

```
** Event #2414.  T=43.184274  (43.18s).  Module #88 `net80211.host[5].rap.logic'

RAPLogic: received signal message
lamport case sending round number 3, M= 2
Taking a decision: beginning step one: set the leaf value
Taking a decision: beginning step two: work up the tree
Taking a decision: beginning step one: set the leaf value
Taking a decision: beginning step two: work up the tree
Process 5 decides on value 1
end lamport, tf=43.1843 total execution time is 27.1831
```

```
** Event #2436.  T=45.090738  (45.09s).  Module #74 `net80211.host[4].rap.logic'

RAPLogic: received signal message
lamport case sending round number 3, M= 2
Taking a decision: beginning step one: set the leaf value
Taking a decision: beginning step two: work up the tree
Taking a decision: beginning step one: set the leaf value
Taking a decision: beginning step two: work up the tree
Process 4 decides on value 1
end lamport, tf=45.0907 total execution time is 29.0896
```

```
** Event #2466.  T=47.610973  (47.61s).  Module #102 `net80211.host[6].rap.logic'

RAPLogic: received signal message
lamport case sending round number 3, M= 2
Taking a decision: beginning step one: set the leaf value
Taking a decision: beginning step two: work up the tree
Taking a decision: beginning step one: set the leaf value
Taking a decision: beginning step two: work up the tree
Process 6 decides on value 1
end lamport, tf=47.611 total execution time is 31.6098
```

```
** Event #2479.  T=48.75067  (48.75s).  Module #32 `net80211.host[1].rap.logic'

RAPLogic: received signal message
lamport case sending round number 3, M= 2
Process 1 is faulty
end lamport, tf=48.7507 total execution time is 32.7495
```

La simulazione proposta ha termine infatti all' evento 3007 , con $\boldsymbol{Tf} \cong 48{,}75\ secondi$, e i tempi di esecuzione medi di ciascun processo, come possiamo vedere, si aggirano tutti sui $30\ secondi$.

CAPITOLO 9

CONCLUSIONI E SVILUPPI FUTURI

Gli algoritmi di consenso sono alla base di una qualunque applicazione reale in un sistema distribuito in cui non si possa prescindere dal considerare l'occorrenza di guasti occasionali o maliziosi. Questi algoritmi sono complessi per loro natura e di volta in volta possono essere sviluppati algoritmi ad hoc per il contesto applicativo in cui andiamo ad operare. Questo consente di ridurre la generalità dell'algoritmo e quindi di ottenere risultati migliori in termini di scambio di messaggi e tempi di esecuzione. In questo lavoro, abbiamo preso in considerazione il protocollo RAPTOR, proposto per risolvere il problema del consenso nel contesto dell'Air Traffic Management. Dalle simulazioni effettuate in nell'ambiente OMNET++ possiamo vedere che, nella maggior parte degli scenari reali, a partire dalle stesse ipotesi, il numero di messaggi totali scambiati durante il protocollo RAPTOR è inferiore rispetto a quelli scambiati applicando l'algoritmo classico dei Generali Bizantini.

Anche i tempi di esecuzione del protocollo ottenuti con la simulazione sono drasticamente maggiori per l'algoritmo dei Generali Bizantini. Una simulazione proposta di due processi faulty, sette host in totale, mostrava un tempo di esecuzione $\boldsymbol{Tf} \cong 48{,}75\ secondi$ per l'algoritmo dei Generali Bizantini, mentre i tempi di esecuzione del protocollo RAPTOR sono dell' ordine del secondo. Queste osservazioni portano ad affermare che la scalabilità del protocollo RAPTOR è maggiore rispetto al protocollo dei Generali Bizantini. I possibili sviluppi futuri sono i seguenti.

9.1 Il modulo SGT.

Abbiamo visto come gli aircraft possono scambiarsi informazioni in modo distribuito tramite l' ADS-B e raggiungere un agreement su queste informazioni in modo del tutto affidabile tramite lo stack di protocolli RAPTOR. Quello che rimarrebbe da analizzare, per una piena e completa realizzabilità pratica nel mondo reale del sistema implementato, è il livello di logica SGT.

Ogni compagnia aerea difatti gestisce, definisce, e implementa le proprie politiche di gestione delle rotte del traffico aereo, secondo i criteri più disparati, e ovviamente ognuna in modo indipendente dalle altre.

La logica che implementa la scelta della rotta sulla base della politica gestionale della compagnia, e sulla base delle informazioni costantemente ricevute e rilanciate via satellite sulle posizioni e sulle rotte degli altri arerei, dovrebbe essere implementata in questo modulo dell'aircraft. La scelta della rotta dovrebbe avvenire sulla base delle informazioni ottenute dal protocollo RAPTOR sottostante, e del calcolo del valore ottimo di una certa funzione obiettivo che definisce la politica scelta dalla compagnia.

Sarebbe interessante sviluppare e implementare sull'SGT alcune delle politiche gestionali di "base", o comunque di maggiore utilizzo e più comuni nel campo della gestione del traffico aereo.

9. 2 Servizi RAPTOR per l' airborne self separation

Per l' effettiva applicabilità dei protocolli in situazioni e ambienti reali, RAPTOR mette a disposizione alcuni servizi: di questi ne illustreremo brevemente alcuni che riguardano da vicino le problematiche

affrontate fin'ora, e di cui sarebbe interessante fare un'analisi delle problematiche associate e della loro implementazione.

9.2.1 Group Membership Service

Il primo servizio è un estensione dello stack di protocolli, che rende il RAPTOR adatto all' ambiente fortemente dinamico su cui si appoggia l' airborne self separation. Nell' illustrare gli algoritmi di consenso binario e multivalore abbiamo implicitamente assunto che il gruppo di processi, cioè di aircraft in situazioni reali, che componevano l' insieme ∏ fosse statico nel tempo.

Questo nella realtà non è ovviamente vero, durante tutto il volo di un determinato aereo, il gruppo di aircraft con i quali vengono scambiate informazioni al fine di raggiungere gli agreement richiesti, cambia nel tempo.

L' assunzione di una conoscenza fissa e a priori del gruppo di processi è sicura soltanto una volta che siamo in prossimità di aereoporti, dove l' infrastuttura di terra può fornire informazioni affidabili sugli aircraft posti nelle vicinanze.

Una volta in volo, questa infrastruttura viene a mancare e gli aircraft necessitano di un modo per poter aggiornare costantemente il gruppo a cui appartengono in un ambiente decentralizzato.

Ogni aircraft deve essere equipaggiato con un aircraft detector, in grado di potere tracciare la presenza di altri aircraft nelle vicinanze in modo eventualmente incompleto (cioè può fallire il detect in determinati istanti). A tale proposito gli aircraft commerciali ad oggi, sono dotati di ricevitori con tecnologia ADS-B.

La soluzione proposta è il *group membership service*.

Questo servizio si basa su un algoritmo che viene eseguito ad intervalli di tempo prefissati e consente a un qualsiasi gruppo esistente di aerei di accordarsi su un set di aircraft che si trovano all' interno di una determinata distanza da uno qualsiasi di loro, e sulla base di questo set, eseguire l' update del loro *group-information.*

Questo presuppone che ogni aircfrat sia inizializzato con un determinato *group-information* iniziale (abbastanza semplice da ottenere, dal momento che ogni aicraft inizia le sue operazioni in aereoporto, la cui struttura di supporto può fornire facilmente queste informazioni).

L' update di un *group-information* può significare aggiungere un nuovo membro, partizionare il gruppo, smantellare il gruppo stesso etc., dipende tutto dai criteri e dalle scelte utilizzate.

Il protocollo è definito come segue:

1. Ciascun aircraft esegue il broadcast delle informazioni locali fornite dal proprio *aircraft detector module* (ossia il set di aircraft visibili da questo a partire dall 'ultima esecuzione dell' algoritmo) a tutti gli aircraft del proprio gruppo ∏.
2. Ciascun aircraft riceve l' informazione di cui al passo 1, da ciascun aircraft del proprio gruppo ∏.
3. L' informazione ricevuta viene memorizzata in un set *DA* (database of aircraft) che contiene tutti gli aircraft osservabili da tutti gli aircraft del gruppo ∏. Si fa notare che la struttura dati *DA* può contenere aircraft del gruppo ∏ e non.
4. La struttura dati *DA* viene passata ad un algoritmo che va a formare uno o piu gruppi in accordo alla posizione geografica degli aircraft presenti in *DA.* Notare che i protocolli RAPTOR per essere eseguiti correttamente necessitano che tutti i processi di uno stesso gruppo ∏ si all'interno del raggio di comunicazione. A seconda dei criteri definiti e utilizzati per la

composizione dei gruppi è possibile che un aircraft possa essere inclusi all' interno di più di un gruppo.

5. L' informazione relativa ai nuovi gruppi viene spedita in broadcast agli aircraft presenti in *DA* che non fanno parte di ∏.
6. Infine, tutti gli aircraft (sia quelli presenti in ∏, sia quelli presenti in *DA* ma non in ∏) eseguono l' update del loro *group-information* in accordo ai nuovi gruppi formati.

A questo punto è possibile utilizzare il RAPTOR anche in sistemi dinamici in cui il gruppo ∏ di processi che partecipano all'agreement cambia nel tempo.

9.2.2 Rank Consistency Service

Le decisioni sulle manovre ed sugli eventuali cambi rotta di aerei vengono prese sulla base del "ranking" ottenuto dal modulo SGT, e costruito sulla base delle informazioni scambiate tra gli aircraft che si trovano all'interno di un certo range.

Se questa informazione non è consistente tra tutti, è possibile che un aircraft costruisca "ranking" contraddittori tra loro che possono portare a conflitti. Questo può accadere ad esempio se un messaggio viene perduto, corrotto, o inviato di proposito in modo contraddittorio.

Il *rank consistency service* assicura che tutti gli aerei costruiscano lo stesso "ranking", in presenza o meno di alcuni malfunzionamenti. Per ottenere questo, si fa ricorso al *multivalued consensus protocol*, che consente di raggiungere il consenso, ancora sotto la condizione di avere malfunzionamenti originati da non piu di f aircraft, con n=3f+1, dove n è il numero totale di aircraft.

Il servizio è piuttosto semplice: gli aircraft eseguono l' algoritmo di consenso multivalore dove i valori proposti non sono nient' altro che i rankings rkg_i ottenuti dallo scambio di messaggi dell' SGT. Se il

valore *R* ritornato dal protocollo di consenso multivalore è tale che *R* != ⊥ allora ogni aircraft usa *R* (è garantito che tutti abbiano lo stesso *R*).

Altrimenti se il valore è un ranking *R* = ⊥, allora ogni aircraft deve fare affidamento al proprio rkg_i , che può essere incosistente con quello utilizzato da qualche altro aircraft.

9.2.3 View Augmentation Service

Come visto in precedena ogni singolo aircraft p_i può appartenente a più di un gruppo in un determinto istante, dipendentemente dalla distribuzione geografica degli aircraft che lo circondano. Mentre per definizione, ogni aircraft conosciuto da p_i è nel range di comunicazione di p_i, non tutti questi aircraft sono nel range di comunicazione l' uno con l' altro. Questo porta tali aircraft ad essere partizionati in più di un gruppo.

Il *view augmentation service* è implementato con lo scopo di espandere la conoscenza di un determinato gruppo ai gruppi adiacenti. Un gruppo si dice adiacente ad un altro se contengono entrambi uno stesso membro pi.

Dal punto di vista dell' airborne self separation, questa informazione può risultare molto utile nel caso in cui si desideri includere un set di aircraft più grande all' interno di una risoluzione di conflitti eseguita dall' SGT.

BIBLIOGRAFIA

[1] Henrique Moniz Alessandra Tedeschi Nuno Ferreira Neves :

"A Distributed Systems Approach to Airborn Self-Separation", in Li Weigang, Alexandre de Barros, Italo Romani de Oliveira *"Computational Models, Software Engineering and Advanced Techonlogies in Air Transportation"*.

[2]*Paolo Masci, Henrique Moniz, Alessandra Tedeschi:*

"Services for Fault-Tolerant Conflict Resolution in Air Traffic Management" in "*RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems",* Novembre 17-19, 2008

[3]*Paolo Masci, Alessandra Tedeschi*:

"Modelling and Evaluation of a Game Theory Approach for Airborne Conflict Resolution in OMNeT++", in "*2009 Second International Conference on Dependability",* 2009, pp.162-165

[4]*Algirdas Avizienis, Jean-Claude Laprie Brian Randell, Carl Landwher:*

"Basic Concepts and Taxonomy of Dependable and Secure Computing" in "*IEEE Transactions on Dependable and Secure Computing",* Marzo 2004

[5]*Leslie Lamport, Robert Shostak, Marshall Pease:*

"The Byzantine General Problem" in *"ACM Transactions on Programming Languages and Systems"*, Volume 4, 1982, pp. 382-401.

[6] *Russel Turpin Brian A. Coan:*

"Extending Binary Byzantine Agreement to Multivalued Byzantine Agreement" in "*Information Processing Letter",* Volume 18, 28 Febbraio 1984, pp 73-76.

[7] *James K. Kuchar and Lee C. Yang:*

"A Review of Conflict Detection and Resolution Modeling Methods" in *"IEEE Transactions on Intelligent Transportation Systems",* Volume 1, 2000, pp 179-189

[8] *Mark Nelson :* http://marknelson.us/2007/07/23/byzantine/

[9] *ADSB Project URL:* http:/www.ads-b.com/home.html

[10] *OMNeT++ project URL***:** http://www.omnetpp.org

RINGRAZIAMENTI

Desidero ringraziare in primo luogo la Professoressa Cinzia Bernardeschi, docente presso la Facoltà di Ingegneria dell'Università degli Studi di Pisa, primo relatore di questo lavoro, per l' aiuto fornito e per l'enorme disponibilità mostrata nella stesura dello stesso.

Sentiti ringraziamenti vanno inoltre all'Ingegnere Paolo Masci, tra i principali ricercatori e sviluppatori, presso la Facoltà di Ingegneria dell'Università degli Studi di Pisa, di servizi e architetture per la risoluzione dei conflitti in ambito ATM, cui si è fatto riferimento nello sviluppo di questo lavoro.

Salvatore Buonaguro